

UNIVERSIDADE FEDERAL DO PARANÁ

RENAN DE SOUZA POLISCIUC

DIAGNÓSTICO DE TRAÇOS MALICIOSOS NO ANDROID UTILIZANDO CHAMADAS  
DE SISTEMA

CURITIBA PR

2020

RENAN DE SOUZA POLISCIUC

DIAGNÓSTICO DE TRAÇOS MALICIOSOS NO ANDROID UTILIZANDO CHAMADAS  
DE SISTEMA

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Informática no Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Luiz Carlos Pessoa Albini.

Coorientador: André Ricardo Abed Grégio, Luis Carlos Erpen de Bona.

CURITIBA PR

2020

CATALOGAÇÃO NA FONTE – SIBI/UFPR

---

P768d

Polisciuc, Renan de Souza

Diagnóstico de traços maliciosos no android utilizando chamadas de sistema [recurso eletrônico]/ Renan de Souza Polisciuc, 2020.

Dissertação (Mestrado) - Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, da Universidade Federal do Paraná - Área de concentração: Ciência da Computação.

Orientador: Luiz Carlos Pessoa Albini.

Coorientador: André Ricardo Abed Grégio, Luis Carlos Erpen de Bona.

1. Ciência da Computação. 2. Android. I. Albini, Luiz Carlos Pessoa. II. Grégio, André Ricardo Abed. III. Bona, Luis Carlos Erpen de. IV. Universidade Federal do Paraná. V. Título.

CDD 005.35

---

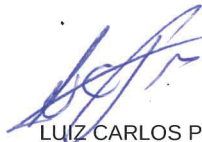
Bibliotecária: Vilma Machado CRB9/1563

## TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da Dissertação de Mestrado de **RENAN DE SOUZA POLISCIUC** intitulada: **Diagnóstico de Traços Maliciosos no Android utilizando Chamadas de Sistema** sob orientação do Prof. Dr. LUIZ CARLOS PESSOA ALBINI, que após terem inquirido o aluno e realizada a avaliação do trabalho, são de parecer pela sua APPROVAÇÃO no rito de defesa.

A outorga do título de mestre está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

CURITIBA, 13 de Março de 2020.



LUIZ CARLOS PESSOA ALBINI

Presidente da Banca Examinadora (UNIVERSIDADE FEDERAL DO PARANÁ)



ANDRÉ RICARDO ABED GRÉGIO

Coorientador - Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)



CARLOS ALBERTO MAZIERO

Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)



JULIANA DE SANTI

Avaliador Externo (UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ)



*A todos aqueles vivem e defendem a  
ciência*

## **AGRADECIMENTOS**

Gostaria de agradecer a meus pais pelo apoio e paciência, minha noiva Bianca por todo o esforço que ela fez para que eu conseguisse concluir este trabalho. Não menos, gostaria de agradecer os professores Luiz Albin, André Grégio e Luis de Bona por me atenderem quando precisei e por se dedicarem ao meu desenvolvimento como cientista da computação.

## RESUMO

O Android é o sistema operacional mais utilizado por dispositivos móveis no mundo. Esse fato tem atraído cada vez mais desenvolvedores para a plataforma devido a sua característica de código aberto e possibilidade de desenvolver aplicações de forma gratuita. Além das aplicações desenvolvidas para melhorar a vida dos usuários e automatizar operações, estão aquelas maliciosas, conhecidas como *malware*, que são desenvolvidas para comprometer dados sensíveis dos usuários e/ou prejudicar o funcionamento de dispositivos. Com base nisso, muitos trabalhos têm surgido para analisar e detectar aplicações maliciosas no Android. Dentre esses trabalhos, estão aqueles baseados em análise estática, que verificam o código da aplicação e aqueles baseados em análise dinâmica, que consiste em executá-la em um ambiente controlado e obter traços da execução. Muitos dos trabalhos baseados em análise dinâmica utilizam as chamadas de sistemas para caracterizar as aplicações, porém focam apenas em verificar a frequência que essas chamadas de sistema ocorrem, o que não é suficiente para tal. As chamadas de sistemas contêm informações relevantes sobre as interações das aplicações com o restante do sistema que são ignoradas pelas soluções. Além disso, muitos ignoram que as características estáticas também podem ser relevantes para a análise. Por isso, neste trabalho apresentaremos o DroiDiagnosis, uma ferramenta que utiliza características estáticas e dinâmicas baseadas em chamadas de sistemas para diagnosticar aplicações no Android.

Palavras-chave: malware, malware android, análise de malware, diagnóstico de aplicativo malicioso

## **ABSTRACT**

Android is the world's most used operating system in mobile devices. This has attracted developers, as it is an open source platform and apps can be developed free of charge. Besides apps that can enhance the life of users and automate tasks, there are the malicious ones, known as malware, which are developed to compromise private user data or cause malfunction to the devices. Taking this into account, several studies are trying to analyze and detect malicious Android applications. From these, some are based on static analysis, that verifies the app source code, and others are based on dynamic analysis, which consists of running the app in a controlled environment and obtaining execution traces. Several studies based on dynamic analysis use the system calls to characterize the app, but focus only on the amount of system calls triggered, which is not enough. System calls contain important information regarding the interaction of the app with the system, that are ignored by the proposed solutions. Moreover, several papers ignore that the static characteristics could also be relevant in the malware analysis. Hence, the DroiDiagnosis will be presented in this study, which uses both static and dynamic features based on system calls to diagnose Android based apps.

**Keywords:** malware, malware android, malware analysis, malicious applications diagnosis



## LISTA DE FIGURAS

2.1	Trecho de código do skygofree para ler conversas do WhatsApp. Disponível em Secure List. . . . .	16
2.2	Adware Cosiloon em ação. Disponível em Android Police. . . . .	17
2.3	Mensagem do DoubleLocker avisando que o dispositivo foi bloqueado. Disponível em We Live Security. . . . .	18
2.4	App que promete instalar 32gb de memória no celular. . . . .	19
2.5	App encontrada em um repositório não oficial.. . . .	19
2.6	DKFBootKit requisitando permissão de root. Disponível em Tech tudo. . . . .	19
2.7	Diferença entre uma APK legítima e sua versão repackaged . . . . .	20
2.8	Processo de análise estática . . . . .	22
2.9	Substituição da SandBox do Android pelo DroidBox . . . . .	24
2.10	Funcionamento do strace . . . . .	24
3.1	Simulação de aplicações e extração de características . . . . .	27
3.2	Arquitetura da aplicação . . . . .	28
3.3	Top 10 - Frequência de chamadas de sistema - Benignas. . . . .	28
3.4	Top 10 - Frequência de chamadas de sistema - Maliciosas . . . . .	29
3.5	Arquitetura da solução . . . . .	29
4.1	Processo de obtenção de traços de uma aplicação . . . . .	32
4.2	Modo de funcionamento de análise. . . . .	34
4.3	Comparação do diagnóstico tradicional e o diagnóstico de traços mínimos . . . . .	35
5.1	Quantidade média de chamadas de sistema por classe de aplicação . . . . .	38
5.2	Tempo médio (em ms) de execução de chamadas de sistema por classe de aplicação	39
5.3	Quantidade média de acessos por diretório de profundidade 0. . . . .	40
5.4	Quantidade média de acessos por diretório de profundidade 1. . . . .	40
5.5	Quantidade média de acessos por diretório de profundidade 2. . . . .	41
5.6	Quantidade média de tentativas de troca de usuário . . . . .	41
5.7	Quantidade média de acessos por domínio . . . . .	42
5.10	Matriz de confusão . . . . .	45
5.11	Matriz de confusão para a análise dinâmica utilizando SVM . . . . .	46
5.12	Matriz de confusão para a análise dinâmica utilizando KNN . . . . .	46
5.13	Matriz de confusão para a análise dinâmica utilizando Decion-Tree. . . . .	47
5.14	Matriz de confusão para a análise estática utilizando SVM . . . . .	47

5.15	Matriz de confusão para a análise estática utilizando KNN . . . . .	47
5.16	Matriz de confusão para a análise estática utilizando Decision-Tree . . . . .	47
5.17	Matriz de confusão para a análise híbrida utilizando SVM. . . . .	48
5.18	Matriz de confusão para a análise híbrida utilizando KNN . . . . .	48
5.19	Matriz de confusão para a análise híbrida utilizando Decision-Tree . . . . .	48
5.20	Fragmentos maliciosos encontrados pela quantidade fragmentos . . . . .	50
5.21	Top 10 chamadas de sistema em 2, 5, 10 e 20 divisões de traços maliciosos. . . .	50
5.22	Top 10 chamadas de sistema em 2, 5, 10 e 20 divisões de traços benignos . . . .	50

## LISTA DE TABELAS

5.1	Métricas de desempenho da análise dinâmica . . . . .	47
5.2	Métricas de desempenho da análise estática . . . . .	48
5.3	Métricas de desempenho da análise híbrida . . . . .	49

## LISTA DE ACRÔNIMOS

ABI	<i>Application Binary Interface</i> (Interface binária do aplicativo)
ADB	<i>Android Debug Bridge</i> (Ponte de depuração do Android)
AOT	<i>Ahead-of-time</i> (Antes do tempo)
API	<i>Application programming interface</i> (Interface de programação)
APK	<i>Android Package</i> (Pacote do Android)
ART	<i>Android Runtime</i> (Tempo de execução do Android)
AVD	<i>Android Virtual Device</i> (Dispositivo virtual do Android)
HAL	<i>Hardware Abstraction Layer</i> (Camada de abstração de hardware)
JIT	<i>Just-In-Time</i> (Na hora certa)
JNI	<i>Java Native Interface</i> (Interface nativa do Java)
JSON	<i>JavaScript Object Notation</i> (Notação de objeto do JavaScript)
KNN	<i>K-Nearest Neighbors</i> (K vizinhos mais próximos)
NFC	<i>Near Field Communication</i> (Comunicação por campo de proximidade)
SDK	<i>Software development kit</i> (Kit de ferramentas de desenvolvimento)
SVM	<i>Support Vector Machine</i> (Máquina de vetores de suporte)
UI	<i>User Interface</i> (Interface do utilizador)
VM	<i>Virtual Machine</i> (Máquina virtual)

## SUMÁRIO

<b>1</b>	<b>INTRODUÇÃO . . . . .</b>	<b>13</b>
<b>2</b>	<b>FUNDAMENTAÇÃO TEÓRICA. . . . .</b>	<b>15</b>
2.1	MALWARE NO ANDROID . . . . .	15
2.1.1	Técnicas de infecção . . . . .	18
2.1.2	Técnicas utilizadas por <i>malware</i> evasivo . . . . .	20
2.2	ANÁLISE E DIAGNÓSTICO DE MALWARE . . . . .	22
2.2.1	Análise Estática . . . . .	22
2.2.2	Análise Dinâmica . . . . .	23
2.2.3	Considerações finais . . . . .	25
<b>3</b>	<b>TRABALHOS RELACIONADOS . . . . .</b>	<b>26</b>
<b>4</b>	<b>DROIDIAGNOSIS. . . . .</b>	<b>31</b>
4.1	COMPONENTES E PROCESSOS. . . . .	31
4.1.1	Obtenção de Características Estáticas . . . . .	31
4.1.2	Obtenção de Características Dinâmicas . . . . .	32
4.1.3	Repositório . . . . .	33
4.1.4	Seleção de Características . . . . .	33
4.1.5	Vetorização . . . . .	33
4.1.6	Classificação . . . . .	34
4.1.7	Detecção de maliciosidade . . . . .	34
4.2	MODO ANÁLISE . . . . .	34
4.3	MODO DIAGNÓSTICO . . . . .	34
<b>5</b>	<b>RESULTADOS. . . . .</b>	<b>36</b>
5.1	BASE DE DADOS . . . . .	36
5.2	EXPERIMENTOS . . . . .	36
5.3	CHAMADAS DE SISTEMA. . . . .	36
5.4	DIRETÓRIOS E ARQUIVOS . . . . .	37
5.5	DADOS DE REDE . . . . .	42
5.6	PERMISSÕES . . . . .	42
5.7	ANÁLISE. . . . .	44
5.7.1	Métricas . . . . .	44
5.7.2	Características. . . . .	46
5.7.3	Base de dados . . . . .	46
5.7.4	Análise dinâmica . . . . .	46
5.7.5	Análise estática . . . . .	47

5.7.6	Análise híbrida . . . . .	48
5.7.7	Comparação entre os modelos . . . . .	49
5.8	TRAÇOS MÍNIMOS . . . . .	49
5.8.1	Quantidade de fragmentos maliciosos . . . . .	49
5.8.2	Quantidade de chamadas de sistemas . . . . .	49
5.9	CONSIDERAÇÕES FINAIS . . . . .	51
5.9.1	Aplicabilidade. . . . .	51
<b>6</b>	<b>CONCLUSÃO E TRABALHOS FUTUROS. . . . .</b>	<b>52</b>
6.1	PROBLEMAS CONHECIDOS. . . . .	52
	<b>REFERÊNCIAS . . . . .</b>	<b>54</b>

## 1 INTRODUÇÃO

O Android é o sistema operacional para dispositivos móveis mais popular do mundo, representando uma parcela de mais de 85% do mercado de smartphones, (Statista, 2018). No Brasil, ele está presente em cerca de 95% dos smartphones, (Lavado, 2019). Essa popularidade tem atraído cada vez mais desenvolvedores para a plataforma, incluindo alguns mal-intencionados que desenvolvem aplicações maliciosas (*malware*) para roubar informações do usuário e/ou danificar o dispositivo. Alguns *malware* podem enviar SMS sem o consentimento do usuário, realizar chamadas telefônicas, roubar dados bancários ou até mesmo bloquear o dispositivo. Com base nisso, muitos trabalhos foram desenvolvidos para analisar aplicações do Android e identificar aquelas que são *malware*.

Trabalhos como DroidMat, (Wu et al., 2012), Drebin, (Arp et al., 2014), e DroidFusion, (Yerima e Sezer, 2018) utilizam a técnica de análise estática, que consiste em descompilar a aplicação e analisar seu código fonte e recursos utilizados para encontrar características que indiquem comportamento malicioso. Para classificar uma aplicação como *malware*, eles utilizam algoritmos de aprendizado de máquina que têm lhes proporcionado taxas de acurácia de mais de 90%. Apesar de possuírem uma alta precisão e serem mais rápidas que outras técnicas, elas têm um problema em comum que é não identificar *malware* que somente se manifeste em tempo de execução, já que sua análise é baseada apenas no código estático. Isso permite que muitos *malware* encriptem strings com comandos e os decifrem quando a aplicação está sendo executada, evadindo assim o *scan*.

A técnica de análise dinâmica é utilizada por trabalhos como DroidScribe, (Dash et al., 2016) e EnDroid, (Feng et al., 2018), e consiste em executar a aplicação para captura de ações maliciosas. A vantagem dessa abordagem é que ela identifica execução de código malicioso ofuscado pelo *malware*. Quanto às desvantagens, elas são mais lentas e complexas que as estáticas, pois exigem que uma VM (*virtual-machine*) seja levantada ou então um dispositivo real seja utilizado. Em ambos ambientes o dispositivo deve ser reiniciado e ter todos os seus dados removidos a cada simulação para evitar influências de uma simulação em outra. Uma desvantagem adicional de simulação em ambientes emulados é que alguns *malware* podem identificar que estão sendo executados nesses ambientes. Outra desvantagem comum da análise dinâmica é que, devido ao tempo limitado de simulação e a aleatoriedade das ações, algumas partes da aplicação podem não ser alcançadas.

Um problema de alguns trabalhos que realizam análise dinâmica é que eles utilizam apenas o rastreamento de ações dentro da *sandbox* das aplicações, enquanto que algumas atividades maliciosas podem ser realizadas fora desse espaço. Com base nisso, alguns trabalhos têm utilizado chamadas de sistemas, que são a forma de troca de informações entre as aplicações e o *kernel* e que podem ser facilmente acessadas através de ferramentas do próprio Linux. Trabalhos como (Burguera et al., 2011), (Malik, 2016), e (Jaiswal et al., 2018) utilizam chamadas de sistema para diagnosticar e classificar aplicativos maliciosos. O problema principal é que eles se baseiam apenas na quantidade de chamadas de sistemas efetuadas pelas aplicações para determinar padrões de classificação, que, como mostrado neste trabalho, são insuficiente para determinar a maliciosidade de uma aplicação.

Sendo assim, neste trabalho introduziremos o DroiDiagnosis, uma ferramenta baseada em chamadas de sistemas e permissões de acesso para diagnosticar aplicações maliciosas. Serão apresentados resultados relacionados a quantidade de chamadas de sistema, diretórios acessados e URLs as quais houve trocas de informações pelas aplicações. Também iremos mostrar

como podemos identificar algumas operações ilegais através de chamadas de sistemas. Por fim utilizaremos todas essas características para classificar as aplicações utilizando os classificadores de aprendizado de máquina SVM, KNN e Decision Tree. Além disso, apresentaremos uma forma otimizada de analisar traços do Android de maneira que os traços possam ser analisados em pequenas partes independentes.

Este trabalho está dividido por: no Capítulo 2 introduziremos os fundamentos de análise de *malware*, em 2.1 os fundamentos de *malware* no Android, em 3 os trabalhos relacionados, seus problemas e características, em 4 apresentaremos o DroiDiagnosis, suas funcionalidades e arquitetura, em 5 os resultados relevantes e concluímos o trabalho no Capítulo 6.



## 2 FUNDAMENTAÇÃO TEÓRICA

Assim como outros tipos de sistemas operacionais, o Android possui aplicativos maliciosos e formas de identificá-los. Por ser um sistema aberto e de desenvolvimento gratuito, o Android atrai muitos desenvolvedores, incluindo aqueles mal-intencionados. Em contraste a isso, o fato de ser de código aberto possibilita que estruturas do sistema sejam estudadas pela comunidade para criação de soluções anti-*malware* através de técnicas já estudadas em outros sistemas operacionais. Sendo assim, neste capítulo apresentaremos os tipos de *malware* encontrados para o Android e as técnicas de análise e detecção encontradas na literatura.

### 2.1 MALWARE NO ANDROID

Os *malware* para Android são aplicações que aproveitam de brechas da API para explorar falhas do sistema e realizar tarefas maliciosas. Os principais objetivos desse tipo de aplicação são roubar informações dos usuários ou danificar o dispositivo, sendo que alguns deles podem utilizar créditos telefônicos para enviar SMS e realizar chamadas, travar o *smartphone* com operações inúteis e até 'sequestrar' o dispositivo, como é o caso do *ransomware*, (Kubovič, 2018).

Apesar da evolução do Android e surgimento de novas técnicas de detecção cada vez mais precisas, os *malware* ainda estão em alta. Em dezembro de 2017, a Kaspersky identificou um *malware* (posteriormente chamado de Loapi) que esquentava tanto o dispositivo (fazendo operações de criptomoedas) que 'estufa' a bateria, (Markovskaya, 2017). A *Avast Threat Labs* identificou um *adware*, *malware* que mostra propagandas desnecessárias e em excesso, pré-instalado em dispositivos não certificados pela Google como ZTE e Archos, (Dent, 2018). A *Researchers at ThreatFabric* descobriu um trojan para Android 7 e 8 nomeado MysteryBot que se disfarça de app bancária para realizar ações maliciosas como chamadas telefônicas, ler contatos da lista telefônica e encriptar arquivos pessoais do usuário, (O'Donnell, 2018).

Com o passar dos anos, os ataques a dispositivos Android foram se diversificando conforme novas falhas foram sendo descobertas. A categoria de um *malware* é definida de acordo com o tipo de ação maliciosa que ele efetua. As categorias de *malware* conhecidas são: *trojan*, *backdoor*, *worm*, *botnet*, *spyware*, *adware* e *ransomware*.

- **Trojan:** são apps maliciosas mascaradas como apps benignas cujo principal objetivo é vaziar informações confidenciais como números de contatos, mensagens e até mesmo senhas do usuário. Até 2012, a maioria dos trojans identificados eram capazes de enviar mensagens SMS para números 'premiados' <sup>1</sup> sem consentimento do usuário, podendo em alguns casos causar danos financeiros. Alguns trojans conhecidos: *Trojan-SMS.AndroidOS.FakeInst.ef*, (Unuchek, 2013), *Sypeng*, (Haase, 2017), e *Skygofree*, (Markovskaya, 2018).
- **Backdoor:** é um tipo de *malware* que burla a segurança do sistema para facilitar a entrada de outros *malware*. Pode explorar falhas para ganhar privilégios de superusuário e se esconder de *anti-malware*. Alguns exemplos de backdoors: *Spade*, (LydeckerBlack, 2016), *GhostCtrl*, (Bermejo, 2017), e *TheFatRat*, (GURUBARAN, 2018).

---

<sup>1</sup>Números premiados são números telefônicos que quando chamados geram um custo adicional a quem está realizando a chamada em benefício do número chamado. Alguns exemplos de números premiados são telefones de suporte técnico e de votação para programas de televisão.

- **Worm:** É um tipo de malware que pode criar cópias de si mesmo e se espalhar para outros dispositivos através da rede ou de mídias removíveis. O *Android.Samsapo*, (Symantec, 2014), é um tipo de *worm* que envia mensagens SMS para todos os contatos do dispositivo e que abre uma *backdoor* para *download* de arquivos maliciosos. Outro exemplo de *worm* para o Android é o *Android.Obad*, (Symantec, 2013), que se espalha através do *bluetooth* e que realiza várias ações maliciosas, como envio de SMS para números 'premiados', roubo de informações e *download* de arquivos.
- **Botnet:** Infecta dispositivos e forma uma rede de *bots* chamada *botnet*, que recebe comandos de um servidor externo para efetuar ações maliciosas de forma massiva. O mais comum é que as *botnets* sejam usadas para ataques DDoS (Distributed Denial of Service), (Rouse, 2017). O *WireX*, (Krebs, 2017), é um *botnet* descoberto em agosto de 2017 cujo objetivo é efetuar pequenos ataques *online* e que atingiu 10 mil dispositivos em apenas 2 semanas. Detectado em agosto de 2016, o *DressCode*, (Snow, 2016), é um *malware* que se conecta a um servidor externo e espera comandos do mesmo. Enquanto não recebe comandos, o *malware* fica em estado *sleep*, o que torna sua detecção praticamente impossível. Ao receber o comando, ele entra em estado *wake up* e transforma o dispositivo em uma espécie de *proxy* para redirecionar tráfego de Internet.
- **Spyware:** A função dos *spywares* é roubar informações do dispositivo, como contatos, mensagens, localização e dados bancários e enviá-los para um servidor externo. O *trojan* já citado, *Skygofree*, também atua como *spyware*, pois possui funcionalidades como gravação de áudios e monitoramento de mensagens do *WhatsApp*, (Buchka, 2017). A Figura 2.1 mostra um trecho de código do *SkygoFree* utilizado para este fim, (Skygofree, 2018). O *Tempted Cedar Spyware* é um *malware* desenvolvido para roubar informações de contatos, logs de chamadas, SMS, fotos e configurações do dispositivo, além de ter o poder de gravar áudios e chamadas em andamento, (Abel, 2018). A característica mais interessante desse *malware* é que ele utiliza a engenharia social para atrair usuários e infectar dispositivos. Ele cria perfis falsos no Facebook usando fotos roubadas de mulheres atraentes para enganar os usuários (na maioria homens) e induzi-los a uma conversa particular onde eles devem baixar uma app chamada *Kik Messenger App*.

```
private static void startParsing452065(AccessibilityEvent event) {
    AccessibilityNodeInfo v1;
    try {
        if(event.getEventType() == 32) {
            v1 = event.getSource();
            if(v1 != null && (event.getClassName().equals("com.whatsapp.Conversation")))
                WaParser.mInside = true;
            WaParser.sender = "";
            WaParser.mCount2 = 0;
            WaParser.mChatType = 0;
            WaParser.getSender(v1);
            return;
        }

        WaParser.mInside = false;
        return;
    }

    if(event.getEventType() != 2048) {
        return;
    }

    v1 = event.getSource();
}
```

Figura 2.1: Trecho de código do skygofree para ler conversas do WhatsApp. Disponível em Secure List.

- **Adware:** Os *adware* são *malware* mais 'inofensivos' cujo objetivo é fazer propagandas de forma agressiva. Alguns *adware* criam atalhos na tela principal do dispositivo, roubam informações de favoritos dos navegadores, mudam a página inicial do navegador e mostram anúncios desnecessários. O *Cosiloon*, (Bocek e Chrysaidos, 2018), é um *adware* que ficou famoso por vir pré-instalado em dispositivos das empresas ZTE e Archos. A Figura 2.2 mostra o Cosiloon exibindo propagandas enquanto o usuário está utilizando o dispositivo.

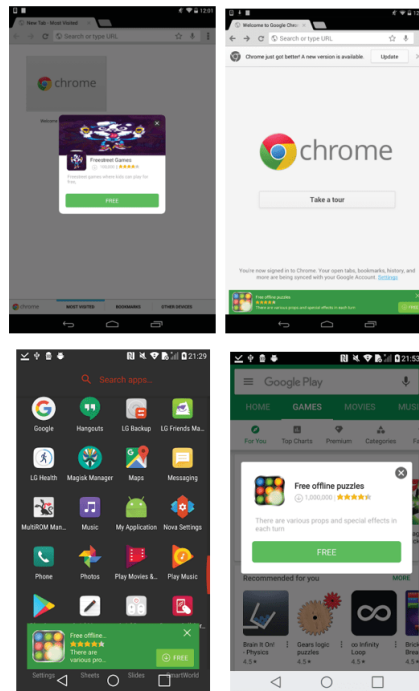


Figura 2.2: Adware Cosiloon em ação. Disponível em Android Police.

- **Ransomware:** O *ransomware* é um malware que pode bloquear o dispositivo e torná-lo inacessível até que uma quantia de dinheiro seja paga. Em 2017 foi descoberto o DoubleLocker, (Kubovič, 2018), um *malware* que é distribuído através de uma falsa versão do *Adobe Flash Player* e que tenta roubar dinheiro de contas bancárias do dispositivo. Além disso, o DoubleLocker altera o código PIN do dispositivo e exige que o usuário efetue o pagamento do resgate. A Figura 2.3 mostra a tela do DoubleLocker no momento em que o dispositivo é bloqueado e o pagamento é exigido.



Figura 2.3: Mensagem do DoubleLocker avisando que o dispositivo foi bloqueado. Disponível em We Live Security.

### 2.1.1 Técnicas de infecção

As técnicas de infecção são meios de os *malware* se instalarem nos dispositivos. A maioria dos *malware* surgem como soluções fantásticas ou então fantasiados de apps legítimas. Acessando o repositório oficial de APKs do Android, encontramos algumas aplicações revolucionárias, como a da Figura 2.4, que promete adicionar 32 GB de memória ram ao dispositivo. Obviamente se trata de um *malware*, e ele foi removido posteriormente do repositório. No entanto, através de uma busca simples na Internet, ele foi encontrado em mais 3 repositórios não oficiais. A Figura 2.5 mostra a app em um desses repositórios.

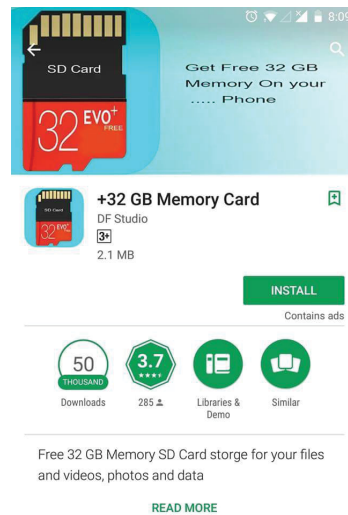


Figura 2.4: App que promete instalar 32gb de memória no celular

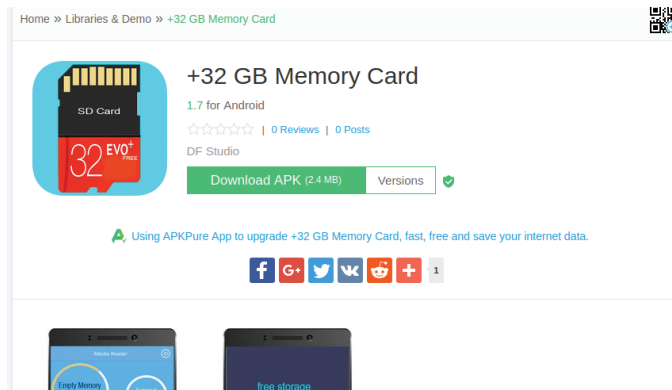


Figura 2.5: App encontrada em um repositório não oficial.

Alguns *malware* como o DKFBootKit, (Kumar, 2018), enganam o usuário de maneira a obter permissão de *root* do sistema. Em 2012, em menos de duas semanas 1.657 aparelhos foram infectados pelo DKFBootKit, sendo que ele foi encontrado em 50 apps diferentes (usando a técnica de *Repackaging*). A Figura 2.6 mostra como o *malware* persuadia o usuário.

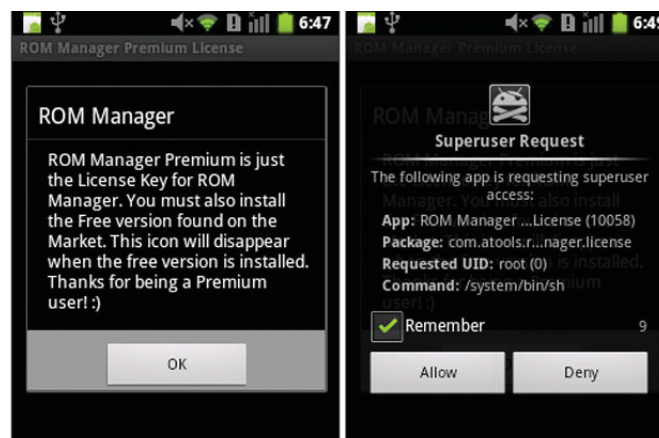


Figura 2.6: DKFBootKit requisitando permissão de root. Disponível em Tech tudo.

O *Repackaging* é uma técnica utilizada pela maioria dos *malware* conhecidos que consiste em descompilar uma aplicação legítima, adicionar código malicioso, compilar e disponibilizá-la em *app-stores* menos monitoradas. A descompilação/recompilação de *apk*s pode ser feita com ferramentas como o *apktool*, (Wisniewski, 2010). Em, (Zhou et al., 2012), é apresentado um exemplo de uma aplicação e sua versão *repackaged*, como mostrado na Figura 2.7. No quadro da esquerda está a versão original e no quadro da direita a versão *repackaged*, onde podemos notar que há uma diferença muito pequena entre ambas, o que pode passar despercebido por alguns softwares *anti-malware*.

Original <i>Angry Birds</i> (in the official Android Market)	Repackaged <i>Angry Birds</i> (in a US alternative marketplace)
<pre>&lt;manifest android:versionCode="142"   android:versionName="1.4.2"   android:installLocation="preferExternal"   package="com.rovio.angrybirds" xmlns:android="....."&gt;    &lt;application android:label=.....&gt;     .....     &lt;meta-data android:name="ADMOB_PUBLISHER_ID"       android:value="a14c9c5b4602e23" /&gt;     &lt;meta-data android:name="ADMOB_INTERSTITIAL_PUBLISHER_ID" android:value="a14ca2471ee0891" /&gt;     .....   &lt;/application&gt; &lt;/manifest&gt;</pre>	<pre>&lt;manifest android:versionCode="142"   android:versionName="1.4.2"   android:installLocation="preferExternal"   package="com.rovio.angrybirds" xmlns:android="....."&gt;    &lt;application android:label=.....&gt;     .....     &lt;meta-data android:name="ADMOB_PUBLISHER_ID"       android:value="a14ce0cb83321d2" /&gt;     &lt;meta-data android:name="ADMOB_INTERSTITIAL_PUBLISHER_ID" android:value="a14ce0cbd3cc9a1" /&gt;     .....   &lt;/application&gt; &lt;/manifest&gt;</pre>

Figura 2.7: Diferença entre uma APK legítima e sua versão *repackaged*

*Drive-by Download* é uma técnica que utiliza engenharia social para persuadir o usuário a baixar uma aplicação maliciosa através de anúncios agressivos. Um exemplo de *malware* que a utiliza é o *Tempted Cedar Spyware*.

A técnica *Dynamic Payload* é utilizada por *malware* que carrega arquivos *.jar/.apk* da pasta *resources* em tempo de execução com objetivos maliciosos. Se o arquivo for do formato *.jar*, é possível utilizar o *DexClassLoader* para carregá-lo. Se for do formato *.apk*, o Android pedirá ao usuário para confirmar se é uma nova atualização do aplicativo. Alguns aplicativos também podem utilizar *Runtime.exec* para executar códigos nativos do Linux de maneira equivalente ao *fork()/exec()*.

### 2.1.2 Técnicas utilizadas por *malware* evasivo

As técnicas utilizadas por *malware* evasivo são estratégias utilizadas pelos *malware* para evadir-se de *software anti-malware*. O que eles fazem é ir na contra-mão dos sistemas de reconhecimento, que capturam suas características e as processam em algoritmos de classificação. Alguns *malware* conseguem se safar apenas aumentando o tamanho dos seus arquivos, o que faz com que soluções baseadas em assinatura não os reconheçam. Outros *malware* executam código malicioso de forma dinâmica para burlar *anti-malware* que analisam o código fonte da aplicação. Alguns outros, como o *DressCode*, que executam esporadicamente, fazem sua detecção seja quase impossível até mesmo por *anti-malware* que analisam em tempo de execução.

As técnicas de furtividade encontradas foram: Inserção de código inútil e Goto, Renomeando Pacotes, Classes e Métodos, Encriptação de strings, Encriptação de recursos, Reflexão de métodos e Identificando emulação.

- **Inserção de código inútil:** Inserção de código inútil aumenta o tamanho da aplicação e muda a *hash* da mesma, o que pode ser usado para enganar *anti-malware* baseado em assinatura.



- **Goto:** *Goto* é um comando que muda a ordem dos comandos sem uma condição lógica, o que pode ser usado para evadir *anti-malware* baseados no fluxo dos dados.
- **Renomeando Pacotes, Classes e Métodos:** Alguns *malware* renomeiam pacotes, classes e métodos para mudar a assinatura da aplicação, fazendo com que *anti-malware* que se baseiam na assinatura sejam enganados.
- **Encriptação de strings:** Strings como mensagens, urls e comandos *shell* podem indicar ações maliciosas de uma app. Para evitar que sistemas de análise utilizem essas informações para detecção, alguns *malware* criptografam essas strings e descriptografam durante a execução da aplicação. Essa técnica é utilizada para evadir análise de *malware* estática.
- **Encriptação de recursos:** Assim como a encriptação de *strings*, recursos da aplicação como código nativo, códigos dex e arquivos .jar podem ser criptografados e utilizados em tempo de execução.
- **Reflexão de métodos:** A reflexão de métodos permite a criação de objetos e invocação de métodos a partir de strings literais em tempo de execução. Essa técnica pode ser detectada com uma análise do fluxo dos dados, porém alguns *malware* podem criptografar as strings, tornando a tarefa mais difícil.
- **Identificando emulação:** um *malware* pode identificar o ambiente o qual está sendo executado e evitar ações maliciosas que podem ser monitoradas por soluções *anti-malware*. O Algoritmo 2.1 apresenta uma forma de identificar execução da aplicação em um *software* de emulação. O Android EmulatorDetector, (Gingo, 2013), é uma biblioteca simples, mas que pode ser usada de forma mais abrangente que o algoritmo mostrado.

```

1 public static boolean isEmulator() {
2     return Build.FINGERPRINT.startsWith("generic")
3         || Build.FINGERPRINT.startsWith("unknown")
4         || Build.MODEL.contains("google_sdk")
5         || Build.MODEL.contains("Emulator")
6         || Build.MODEL.contains("Android SDK built for x86")
7         || Build.MANUFACTURER.contains("Genymotion")
8         || (Build.BRAND.startsWith("generic")
9             && Build.DEVICE.startsWith("generic"))
10        || "google_sdk".equals(Build.PRODUCT);
11 }

```

Listing 2.1: Identificando emulação no Android

As técnicas de invasão e furtividade apresentadas são utilizadas por vários *malware* encontrados na literatura. Verificamos que inevitavelmente a infecção de dispositivos por *malware* é em grande parte por ingenuidade dos usuários e, por mais que o sistema seja robusto e seguro, haverá brechas que poderão ser exploradas por um atacante. Sabendo disso, muitos trabalhos têm investigado formas de detectar apps maliciosas através da identificação das técnicas mencionadas.

## 2.2 ANÁLISE E DIAGNÓSTICO DE MALWARE

A análise e o diagnóstico de aplicações maliciosas pode envolver processos de reengenharia de código ou simulação em dispositivos virtuais ou reais. As técnicas utilizadas para tal podem ser divididas em **estáticas**, que compreendem análise de código, e **dinâmicas**, que envolvem a execução da aplicação e coleta de traços.

### 2.2.1 Análise Estática

A análise estática consiste em verificar o arquivo executável de uma aplicação suspeita e identificar padrões que possam indicar a presença de trechos de código maliciosos. Para que o executável possa ser analisado, é necessário que seja feita a engenharia reversa do seu processo de compilação. No Android, o arquivo executável é em formato .apk, que nada mais é que um arquivo compactado com imagens, bibliotecas, arquivos Java compilados e arquivos de configuração. Para realizar engenharia reversa sob arquivos .apk, algumas ferramentas como a Androguard, (Anthony Desnos, 2018), podem ser utilizadas. A Figura 2.8 apresenta de maneira resumida o processo de análise estática.

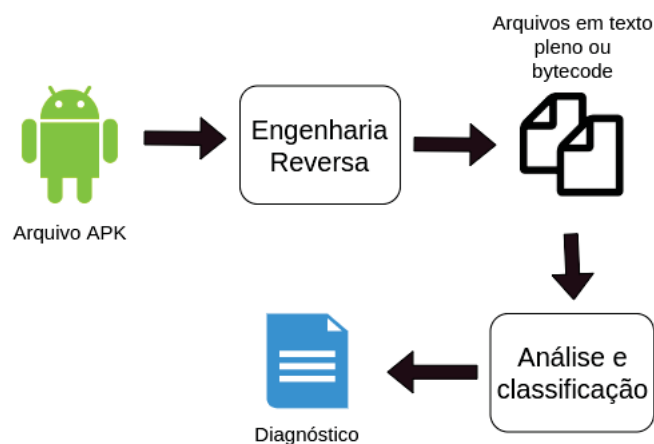


Figura 2.8: Processo de análise estática

Uma das maneiras mais comuns de análise estática é a baseada em assinaturas, que é a mais utilizada por *softwares anti-malware* comerciais, (Faruki et al., 2015). A análise de assinaturas necessita de uma base inicial de assinaturas coletadas de *malware* conhecidos. A partir disso, é possível obter assinaturas das aplicações suspeitas e comparar com o banco de dados procurando por similaridades, que indiquem que a aplicação em questão é semelhante a algum *malware* identificado anteriormente. Algumas soluções como AndroSimilar (Faruki et al., 2013) e DroidAnalytics, (Zheng et al., 2013), também utilizam essa estratégia.

O maior problema da análise usando assinaturas é que como elas dependem de uma base de dados, elas são pouco eficientes em identificar novas variações de *malware* não conhecidas. Com base nisso, alguns trabalhos utilizam outras formas de análise estática. Uma dessas formas é a baseada em permissões requisitadas. As permissões são configurações declaradas no arquivo manifesto da aplicação, (ManifestoAndroid, 2018), em tempo de desenvolvimento, e representam todos os recursos que serão e poderão ser utilizados. Através das permissões podemos prever algumas ações da aplicação, como por exemplo o envio de SMS, realização de chamadas telefônicas, etc.



Alguns trabalhos como (Şahin et al., 2018), (Utku et al., 2018), (Kural et al., 2019) entre outros utilizam a análise de permissões aliada a algoritmos aprendizado de máquina para classificação das amostras. Apesar de parecer suficiente para identificar quais ações uma aplicação pode realizar, muitos desenvolvedores podem utilizar as permissões de maneira incorreta, declarando permissões que não irão utilizar. Em contrapartida, as aplicações maliciosas também podem declarar permissões desnecessárias justamente para confundir os algoritmos de classificação, gerando falsos-positivos ou falso-negativos.

Para aumentar o grau de confiabilidade da solução e tentar diminuir o número de erros das soluções baseadas em permissões, alguns autores introduziram soluções baseadas em chamadas da API. Ao identificar quais são funcionalidades utilizadas pela aplicação, é possível traçar os caminhos possíveis de execução e prever caminhos maliciosos. (Wu et al., 2012) utiliza um grafo de chamadas de API para identificar fluxos de dados maliciosos. Em (Xu et al., 2016) o autor utiliza *Broadcast Receivers*, que são funções de retorno de eventos do sistema (ex: chegada de um SMS), para identificar possíveis roubos de dados sensíveis. Por exemplo, uma aplicação pode esperar a ocorrência da chegada de um SMS no dispositivo e enviar seu conteúdo pela rede. Em (Wu et al., 2017) o autor utiliza *Intents*, que são objetos de comunicação entre aplicações, para identificar vulnerabilidades. Utilizando *Intents*, alguns tipos de *malware* podem realizar ataques em conluio, (Memon e Anwar, 2015).

Alguns trabalhos como (Arp et al., 2014), (Wu et al., 2012) e (Yerima e Sezer, 2018) utilizam tanto permissões como elementos da API, bibliotecas utilizadas e URLs encontradas no código para prever a maliciosidade de uma aplicação. Esses trabalhos coletam dados das aplicações e criam entradas para algoritmos de classificação de aprendizado de máquina. As entradas dos algoritmos são vetores de números inteiros representando as características da aplicação, sendo que cada posição pode ser 0 ou 1, onde 0 é a ausência da característica e 1 a presença.

No geral, as soluções estáticas possuem boa taxa de precisão na identificação de algumas classes de *malware*, são facilmente burladas por encriptação de código. Isso ocorre porque os algoritmos de *parsing* falham ao tentar identificar o código criptografado por não conseguir decifrá-lo. Dessa maneira, a análise estática é pouco efetiva ao tentar diagnosticar aplicações maliciosas que utilizam encriptação de código.

## 2.2.2 Análise Dinâmica

Para contornar os problemas da análise estática, muitos trabalhos utilizam a análise dinâmica, que consiste em simular a execução do *software* em um ambiente isolado e controlado. Esse ambiente pode ser um dispositivo virtual ou real que fornece todos os recursos necessários para que o *malware* realize ações maliciosas. No Android, o ambiente virtual pode ser um emulador, como o oficial do Android, (AndroidStudio, 2019), ou o Genymotion, (Genymotion, 2020). Quanto o ambiente real, pode ser qualquer dispositivo contendo Android, como um *smartphone*.

Existem algumas formas de se analisar aplicações em tempo de execução. A mais comum delas é utilizar uma imagem do sistema operacional modificada para registrar todas as operações que ocorreram no sistema a partir de uma aplicação específica. A ferramenta mais utilizada para este fim é a DroidBox, (pjplantz, 2017), que possui uma imagem do Android 4.1.2 para o *smartphone* Nexus 4 e rastreia várias ações realizadas por uma aplicação, como chamadas telefônicas, envio de SMS, operações criptográficas, etc. De maneira simplificada, o DroidBox substitui a Sandbox padrão do Android, (Bläsing et al., 2010), para interceptar as chamadas enviadas ao sistema operacional a partir da aplicação e gerar relatórios da execução. A Figura 2.9 representa o funcionamento do DroidBox.

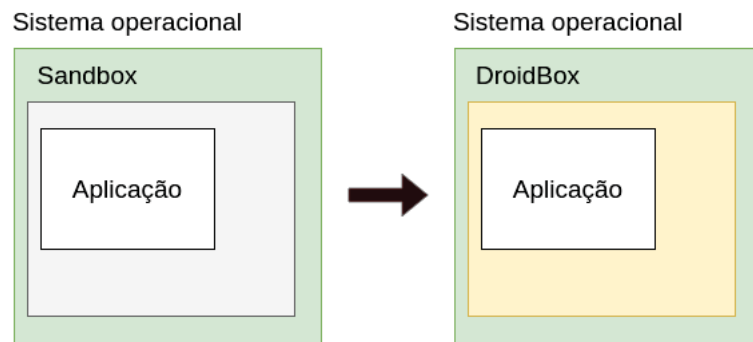


Figura 2.9: Substituição da SandBox do Android pelo DroidBox

Para que o DroidBox substitua a *sandbox* do Android, é necessário que o código fonte do sistema operacional seja recompilado com as alterações necessárias. O código fonte do Android pode ser encontrado na página da AOSP, (AndroidSourceProject, 2016).

Em (Alazab et al., 2012) o autor utilizou o DroidBox para coletar informações de aplicações e verificou que, como ela rastreia apenas a utilização da API do Android, algumas operações que ocorrem em código nativo (bibliotecas compiladas) podem não ser identificadas pelo Droidbox.

Outra forma de analisar aplicações em tempo de execução é utilizando a ferramenta *Strace*, (Strace, 2019). Alguns trabalhos como (Jaiswal et al., 2018) e (Bhatia e Kaushal, 2017) a utilizam. A principal vantagem da utilização dessa ferramenta é que ela intercepta as chamadas efetuadas para o sistema operacional pela aplicação através da API ou fora dela, através de bibliotecas compiladas. Outra vantagem é que o *strace* é já vem nativamente com o Android, portanto não é necessário recompilar o código fonte do sistema para que ela funcione. O funcionamento do *strace* pode ser exemplificado pela Figura 2.10.

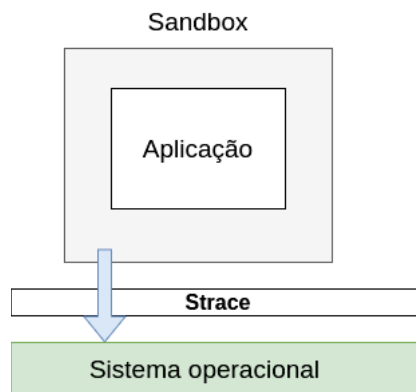


Figura 2.10: Funcionamento do strace

Em (Zaman et al., 2015) o autor apresenta uma solução para detecção de *malware* no Android utilizando análise do tráfego de dados trocados entre o dispositivo e a Internet durante o tempo de simulação. Para que isso fosse possível, ferramentas como TCPDump, Wireshark e *netstat* foram utilizadas. O emulador do Android possui suporte para extração de arquivos .pcap, além disso é possível utilizar o *netstat* dentro do Android para identificar quais conexões estão associadas a cada processo do sistema e assim descobrir qual aplicação está se comunicando.

Um porém da análise dinâmica é que ela leva mais tempo que a estática para dar um diagnóstico e também utiliza mais recursos, pois precisa de uma máquina virtual para executar ou um dispositivo real. Além disso, é necessário que o dispositivo que executa as aplicações esteja com acesso *root*, o que em dispositivos reais pode causar perda do sistema do dispositivo.

### 2.2.3 Considerações finais

Tanto a análise dinâmica quanto a análise estática possuem vantagens e desvantagens. De uma perspectiva quantitativa, a análise estática parece conter menos desvantagens do que a dinâmica, no entanto devido a facilidade com que a estática é burlada, a dinâmica acaba sendo mais vantajosa. Além disso, a análise dinâmica garante uma resposta mais precisa quanto à realização de atividade maliciosas por uma aplicação por ter executado-a. A análise dinâmica pode ser aplicada em ambiente emulado ou ambiente real, sendo que no emulado as aplicações podem detectar que se trata de uma simulação e abortar ações maliciosas. No entanto, em ambientes reais pode ser muito custoso apagar todos os dados do dispositivo a cada simulação, além de correr o risco de perder o dispositivo devido a uma falha no processo.

### 3 TRABALHOS RELACIONADOS

Na literatura são encontrados trabalhos que propõem diagnóstico de *malware* utilizando chamadas de sistema em vários sistemas operacionais. Em (Asmitha e Vinod, 2014) o autor introduz uma solução baseada em chamadas de sistema para detecção de *malware* no Linux. Ele coletou 226 amostras de exemplares maliciosos e 442 benignos e utilizou aprendizado de máquina para classificação. Como características das amostras, o autor utilizou o Strace para coletar as chamadas de sistemas utilizadas por cada processo. As características consistiam de chamadas que só ocorriam em uma das classes e chamadas que ocorriam em ambas. Este trabalho obteve uma acurácia de 97% das amostras classificadas corretamente.

Em (Kolbitsch et al., 2009) o autor introduz um sistema para detecção de *malware* no windows baseado em grafos de chamadas de sistema que se mostrou mais efetiva que as soluções tradicionais baseadas em assinatura. Para obtenção das chamadas de sistema, a solução utiliza o NTrace, que é um equivalente do Strace para o Windows. Uma limitação dessa estratégia é que uma mudança no algoritmo pode gerar um grafo diferente que não foi previsto pela solução. Em (Pham et al., 2019) é apresentado uma ferramenta para auxiliar na extração de características de programas no MAC. A ferramenta permite que características estáticas e dinâmicas sejam extraídas. Ela analisa o comportamento das aplicações através de chamadas de sistema.

A utilização de chamadas de sistema para detecção de *malware* em dispositivos não se restringe apenas a programas utilizados por usuários finais. Em (Tran et al., 2017) o autor introduz um kit de ferramentas para detecção de atividade maliciosa em roteadores comerciais. A principal contribuição deste trabalho é uma *sandbox* que possibilita a extração de chamadas de sistemas executadas e detecção de vulnerabilidades. Para isso, ele utiliza as ferramentas Metasploit e o Strace.

Alguns trabalhos também utilizam as chamadas de sistema para detecção de *malware* no Android. Em (Ahsan-Ul-Haque et al., 2018) introduz um sistema para detecção de aplicativos maliciosos no Android baseado no modelo de Markov. A ideia chave deste trabalho é identificar as transições entre as chamadas de sistemas executadas e a probabilidade da ocorrência de uma chamada de sistema a partir de outra. A base de dados utilizada foi 1.215 aplicações maliciosas e 319 benignas. Usando a técnica de K-Fold e o classificador Naive Bayes, o classificador obteve 98% de acurácia na classificação das amostras.

Inicialmente, são coletados as chamadas de sistemas das aplicações da base de dados que é composta de *malware* e não-*malware*. Após isso, as chamadas de sistemas são colocadas em ordem de ocorrência e é construído um grafo, em que cada nó é uma chamada de sistema e a aresta direcionada de uma para o outro indica que uma chamada de sistema antecedeu a outra. As arestas possuem pesos que indicam a probabilidade de uma determinada chamada anteceder a outra. Através do grafo gerado, é possível extrair uma matriz de características de uma aplicação. Para diminuir o tempo de treinamento dos classificadores, é utilizado *Gaussian Dissimilarity* para seleção das características. A simulação e execução das aplicações são mostrados na Figura 3.1, onde cada aplicação é executada por 10 segundos e seus traços são extraídos utilizando o *strace*.

Entre os pontos negativos desse trabalho é que o tempo para extração do grafo, por aplicação, pode ser um problema, pois é necessário verificar as dependências entre todas as chamadas, o que gera uma complexidade quadrática. Além disso, a base de dados é desbalanceada, sendo que a quantidade de aplicações maliciosas é quase 4 vezes maior que a base de aplicações benignas, o que pode gerar um modelo viciado.

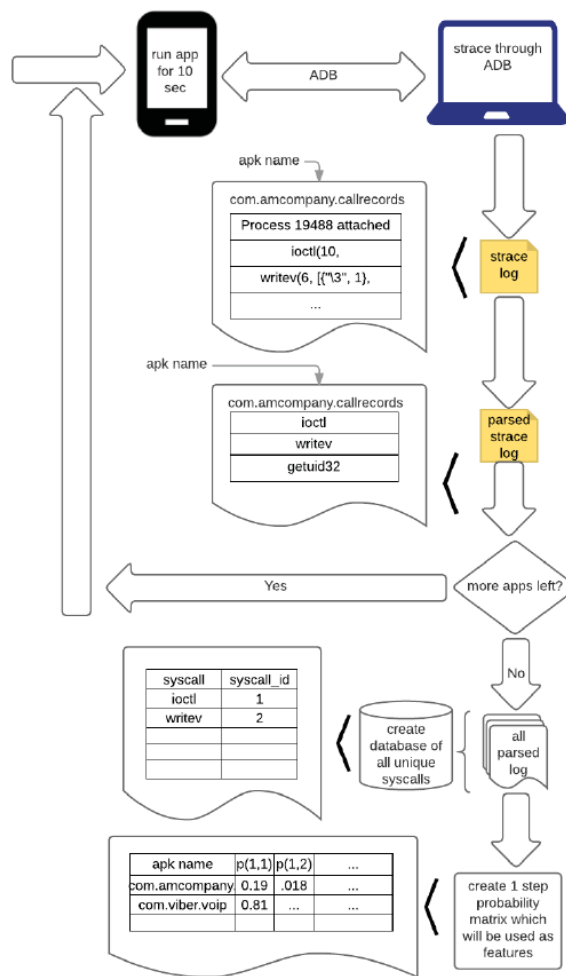


Figura 3.1: Simulação de aplicações e extração de características

Em (Jaiswal et al., 2018) o autor utiliza as chamadas de sistema para analisar o comportamento de jogos e suas versões maliciosas para o Android. A ideia foi contar as quantidades de chamadas de sistemas executadas e utilizá-las para diferenciar as aplicações benignas das maliciosas. A base de dados utilizada continha 20 aplicações benignas e 40 maliciosas. Para analisar as aplicações, a solução proposta é atrelada ao aplicativo no momento em que ele é iniciado pelo sistema, e coleta suas chamadas de sistema até o fim da execução.

Um dos resultados obtidos pelo trabalho, é que aplicativos benignos e maliciosos apresentam as mesmas chamadas de sistemas, mas em frequências diferentes. Outro resultado apresentado é que aplicativos maliciosos requisitam permissões do usuário com o objetivo de explorar o dispositivo e consumir mais recursos do que precisam. Por fim, notou-se que as chamadas de sistemas podem ser utilizadas para diferenciar uma aplicação benigna de sua versão maliciosa, além do que as aplicações maliciosas executam chamadas de sistema com mais frequência que as benignas.

Em (Bhatia e Kaushal, 2017) o autor apresenta uma solução para classificação de aplicações no Android utilizando aprendizado de máquina. Como característica das amostras, o autor utiliza a quantidade de chamadas de sistema executadas por cada aplicação. A base de dados utilizada pelo trabalho era de 50 aplicações benignas e 50 maliciosas, sendo que 85 dessas 100 aplicações foram corretamente classificadas utilizando o método proposto.

A arquitetura da solução é mostrada na Figura 3.2. A arquitetura é composta por 5 etapas que constituem a análise dinâmica. Na primeira etapa, o *framework* de análise é iniciado; na segunda etapa, as aplicações do Android são obtidas; na etapa 3, as aplicações são instaladas e executadas; na etapa 4, os traços de chamadas de sistemas são coletados e, por fim, em 5 a análise é realizada.

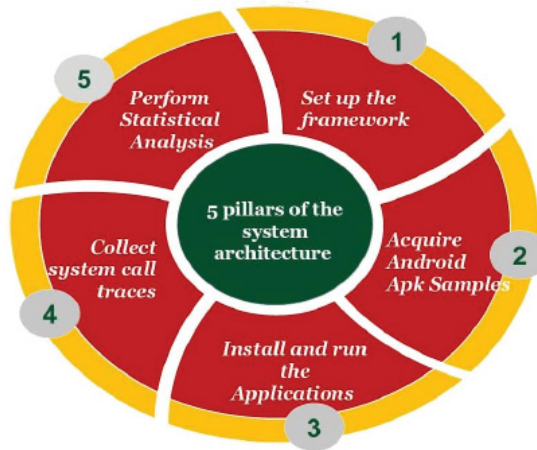


Figura 3.2: Arquitetura da aplicação

Dentre os resultados obtidos por esse trabalho, destacamos as 10 chamadas de sistemas executadas por aplicações benignas e maliciosas e as suas frequências, que podem ser observadas nas Figuras 3.3 e 3.4, que mostram que as frequências de chamadas podem ser usadas para diferenciar as classes de aplicativos.

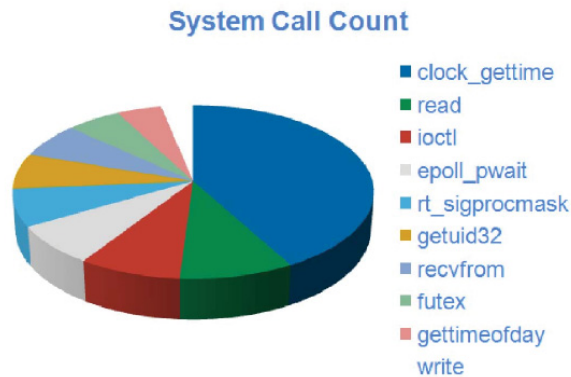


Figura 3.3: Top 10 - Frequência de chamadas de sistema - Benignas

Em (Hou et al., 2016) é apresentado uma solução para análise de *malware* no Android baseado nos grafos de interações entre chamadas de sistemas executadas por uma aplicação. A ideia é que cada nó do grafo é uma chamada de sistema e cada aresta de um nó para outro significa que o nó de origem veio antes do nó de destino. As arestas contêm pesos, que indicam quantas vezes *B* ocorreu depois de *A*. A conclusão apresentada por este trabalho é que a estratégia apresentada é mais precisa que apenas contar as chamadas de sistema executadas. Usando K-fold e *Deep Learning*, em uma base com 3.000 aplicações (1.500 benignas e 1.500 maliciosas), o trabalho obteve precisão de 93.68% das amostras classificadas corretamente. A arquitetura dessa solução é apresentada pela Figura 3.5.



Figura 3.4: Top 10 - Frequência de chamadas de sistema - Maliciosas

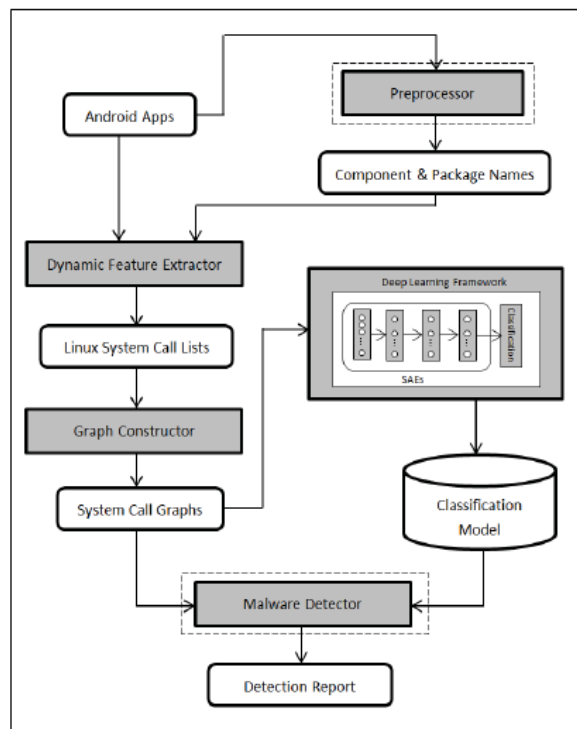


Figura 3.5: Arquitetura da solução

Inicialmente, é feito um pré-processamento da aplicação para poder mapear quais partes da aplicação podem ser atingidas (através da lista de componentes). Em seguida, a aplicação é executada em um ambiente virtual e suas chamadas de sistema são extraídas. Para cada uma das aplicações, é construído um grafo direcionado com pesos, sendo que cada vértice é uma chamada de sistema com um peso baseado na sua frequência, e cada aresta representa uma transição entre chamadas, sendo que o peso das arestas é baseado na frequência da transição. Na sequência é utilizado um algoritmo de *deep learning* sob o grafo gerado para predizer a qual classe a aplicação pertence. No geral, o principal problema dessa solução é que ela é muito custosa devido ao fato de precisar montar um grafo para cada aplicação, que é uma operação quadrática e utilizar *deep learning*, que exige uma grande quantidade de amostras para gerar um modelo consistente.

Nos trabalhos baseados em chamadas de sistema encontrados na literatura, pudemos notar que muitos deles se atentam apenas à quantificação das chamadas e a relação entre elas, mas não à informação contida nas chamadas. Além disso, muitas das soluções que utilizam aprendizado de máquina utilizam bases de dados desbalanceadas ou pequenas, que podem gerar



a falsa impressão de que o método proposto é efetivo. Acreditamos que as chamadas de sistemas, somente, não são suficientes para identificar que uma aplicação é maliciosa. Por isso, neste trabalho iremos explorar mais a fundo as chamadas de sistema no Android e combiná-las com características estáticas. Nosso objetivo é verificar o quão relevante essas características são para um modelo de aprendizado de máquina e que as chamadas de sistemas podem conter informações mais valiosas.

Outro ponto deste trabalho é que como a geração de traços de processos utiliza muitos recursos e, devido às limitações energéticas dos dispositivos móveis, iremos propor uma forma de diagnosticar maliciosidade através de traços mínimos, visando evitar um consumo exagerado de recursos.



## 4 DROIDIAGNOSIS

Neste capítulo introduziremos o DroiDiagnosis, uma solução híbrida para diagnóstico de aplicações maliciosas no Android. A análise híbrida é uma junção da estática e da dinâmica que visa melhorar a precisão em relação às duas individualmente. Alguns trabalhos como (Arshad et al., 2018), (Lindorfer et al., 2014) e (Mas’ud et al., 2013) utilizam a análise híbrida como solução, mas deixam em aberto algumas questões como: a solução híbrida é realmente melhor que as outras duas soluções individualmente? Algumas outras adicionam *overhead* demasiado, como análise em três níveis (estática, dinâmica e rede) em que uma etapa é executada após a outra.

Outra questão não verificada por trabalhos relacionados é se é necessário um traço completo da aplicação para diagnóstico de maliciosidade. A extração de traços de processos é muito custosa e pode comprometer o desempenho do dispositivo. Para isso, o DroiDiagnosis permite o diagnóstico de traços parciais de execução no Android.

Com relação aos trabalhos relacionado, este trabalho se diferencia por propor a análise híbrida utilizando chamadas de sistemas e permissões, que são características simples de serem extraídas. As principais contribuições do DroidDiagnosis são:

- Utiliza características simples de serem extraídas, o que exige menos *overhead*
- Explora elementos das chamadas de sistemas não exploradas por outros trabalhos
- Diagnóstico a partir de traços totais e parciais de execução

Assim como muitos dos trabalhos da literatura optamos por utilizar aprendizado de máquina pois esse método tem se mostrado eficiente na classificação de aplicações maliciosas no Android, além de permitir relacionar características estáticas e dinâmicas e por ser mais flexível para mudanças e melhorias futuras.

### 4.1 COMPONENTES E PROCESSOS

Nesta seção serão apresentados os componentes do DroiDiagnosis e a interação deles para atingir os objetivos propostos. O DroiDiagnosis possui dois modos de operação: análise e diagnóstico. No modo análise são extraídas características estáticas e dinâmicas da aplicação que são processadas de maneira a classificá-la como benigna ou maliciosa.

No modo diagnóstico, apenas a análise dinâmica ocorre e, nesse caso, o objetivo é indicar se o traço processado possui atividades maliciosas. Nesse modo, tanto traços completos como traços parciais podem ser processados. O mecanismo de identificação de atividade maliciosa é o mesmo utilizado para obtenção de características maliciosas do modo análise.

#### 4.1.1 Obtenção de Características Estáticas

O processo de obtenção de características estáticas ocorre em paralelo ao de características dinâmicas e utiliza a ferramenta Androguard, (Anthony Desnos, 2018), para extrair os metadados da aplicação a ser analisada. O Androguard é uma biblioteca do Python que realiza engenharia reversa em arquivos de aplicações do Android e possibilita que diversas informações sejam extraídas, como dados do manifesto, *strings*, classes, métodos, etc. Criamos um serviço cuja entrada é o *sha256* do arquivo da aplicação e consultamos em um repositório de arquivos

.apk para poder enviar para o Androguard. Utilizamos o *sha256* para comprovar que a aplicação não foi alterada de alguma forma e para catalogar as aplicações mais facilmente.

As características estáticas extraídas consistem de permissões e tipos de permissões utilizadas pela aplicação. As permissões escolhidas são as perigosas, que são aquelas que podem afetar a privacidade do usuário e/ou comprometer o funcionamento do dispositivo. Com relação aos tipos de permissões, são consideradas todas as permissões requisitadas pelo dispositivo, sendo elas perigosas ou não.

#### 4.1.2 Obtenção de Características Dinâmicas

O processo de extração de características dinâmicas se inicia com o recebimento do *sha256* pelo serviço de extração. O serviço se encarrega de verificar a existência do arquivo .apk da aplicação no repositório e iniciar uma instância do emulador do Android, (AndroidStudio, 2019), para simulação da execução. Após *n* segundos de simulação, o serviço é finalizado e o processo que o iniciou copia o arquivo do traço de execução, que está em uma pasta interna do dispositivo, para pasta onde ele será processado, fora do dispositivo, onde ele é convertido para JSON e rearmazenado. Além disso, é gerado um arquivo .pcap através do TCPDump que realiza a tarefa de monitorar e extrair os dados de redes que entraram e saíram do dispositivo durante a simulação.

A simulação da execução da aplicação consiste de uma sequência de passos desde preparar o ambiente até gerar comandos pseudo-aleatórios para o dispositivo. 1) Após a aplicação ser encontrada no repositório, utilizamos a ferramenta AAPT para obter o ponto de entrada da aplicação (o seu "main"); 2) Iniciamos uma instância do emulador do Android sem interface gráfica e sem nenhum aplicativo não-padrão em execução, além de associar o TCPDump para capturar os pacotes de rede; 3) Através da ferramenta ADB, (ADB, 2019), instalamos a aplicação no emulador; 4) Após isso iniciamos a aplicação utilizando a informação obtida em 1 sem executar nenhuma ação adicional; 5) Obtemos o *process id* (PID) pertencente à aplicação; 6) Iniciamos o Strace e atrelamos ao PID da aplicação; 7) Iniciamos uma *thread* para geração de toques aleatórios na tela do dispositivo simulando a utilização do mesmo com a ferramenta Monkey; 8) Iniciamos mais um *thread* para enviar comandos Telnet para o dispositivo para simular o envio/recebimento de mensagens de texto (SMS), envio/recebimento de chamadas telefônicas e rotações do dispositivo. Esse processo pode ser ilustrado pela Figura 4.1.

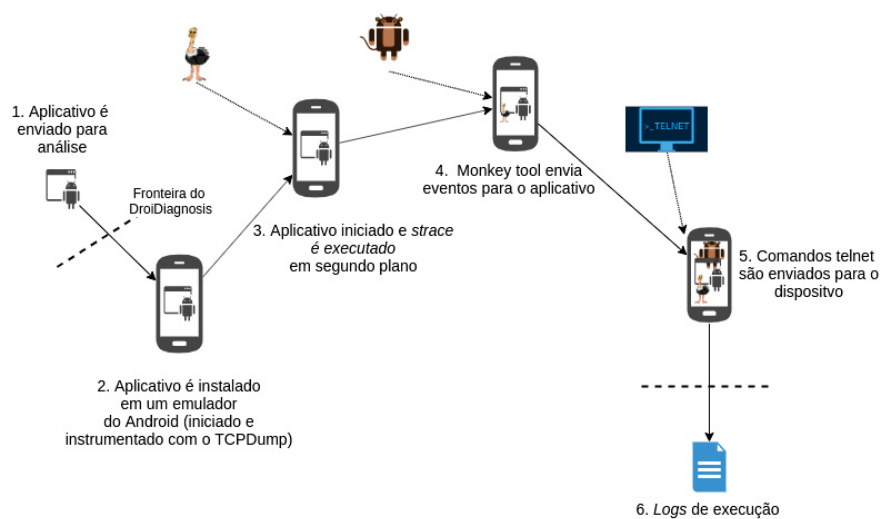


Figura 4.1: Processo de obtenção de traços de uma aplicação

Muitos dos trabalhos na literatura simulam apenas ações dos usuários com toques na tela com o Monkey, mas como apresentado no Capítulo 2.1, alguns tipos de *malware* podem realizar ações maliciosas com a ocorrência de eventos paralelos à aplicação, como o recebimento de SMS. Com base nisso, utilizamos a ferramenta ADB para abrir um *socket* no dispositivo e o Telnet para conectar e enviar comandos simulando a ocorrência de eventos.

Após a obtenção dos traços de execução são extraídas características como: quantidade de chamadas de sistemas por chamadas de sistema, quais chamadas ocorreram, ocorrências de domínios de sites com propaganda, principais diretórios e arquivos acessados, assim como ocorrência de chamadas da função *su* e comandos suspeitos, como *chmod* e *chown* (esses comandos implicam em mais chamadas de sistemas em processos iniciados a partir da aplicação, mas que não são observados).

#### 4.1.3 Repositório

O repositório de aplicativos é um serviço cuja função é buscar arquivos APK e metadados de aplicações em um diretório de arquivos. A comunicação com esse serviço é feita utilizando chamada HTTP informando o *sha256* da aplicação. O retorno do serviço é um arquivo em formato JSON com um vetor de *bytes*, contendo o conteúdo do arquivo APK e informações como classe (benigna ou maligna), tamanho em *bytes*, data de última verificação no VirusTotal, (VirusTotal, 2019), (essa informação é encontrada no AndroZoo, (Allix et al., 2016)), entre outras informações. A maioria dessas informações só é necessária no modo análise durante a fase de treinamento do modelo de aprendizado de máquina, quando precisamos saber as classes das amostras.

#### 4.1.4 Seleção de Características

O processo de seleção de características é executado previamente, de maneira que durante o processo de análise ele seja apenas consultado para filtrar as características. Esse processo consiste em eliminar chamadas de sistemas que foram identificadas como comuns e que não servem para diferenciar aplicações benignas de maliciosas. Assim como as chamadas de sistemas, usando o mesmo critério, também são removidas permissões que ocorrem muitas vezes tanto em uma classe quanto em outra. Diretórios exclusivos das aplicações, como pastas internas e arquivos exclusivos também são removidos. URLs que não são maliciosas também são removidas, sendo que nesse caso, assumimos que as URLs maliciosas são aquelas encontradas em *blacklists* na internet, ou que são conhecidas por possuir propagandas (*adware*).

#### 4.1.5 Vetorização

A vetorização consiste em analisar as características da aplicação e criar um vetor  $V$  de números. Em  $V$ , as características estáticas são as primeiras posições, sendo que cada posição pode ser 0, quando a característica não ocorre na aplicação e 1 quando ocorre. No casos das características dinâmicas, elas são as últimas posições do vetor, sendo que cada posição é a quantidade de vezes que uma certa chamada de sistema ocorreu, a ocorrência ou não de chamadas maliciosas, representadas por 0 ou 1, diretórios restritos, representados por 0 ou 1 quando acessados e a quantidade de vezes que cada diretório foi acessado. A última posição do vetor é a quantidade de URLs maliciosas acessadas. O processo de vetorização só ocorre no modo análise.

#### 4.1.6 Classificação

O processo de classificação consiste em um processo de aprendizado de máquina em que os vetores  $V$  de amostras conhecidas e corretamente classificadas são utilizadas como modelo para classificação de novas amostras. Neste trabalho utilizaremos três classificadores: *SVM*, (RAY, 2017), *KNN*, (SRIVASTAVA, 2018), e *Decision-Tree*, (Gupta, 2017), pois foram os mais utilizados pelos trabalhos encontrados na literatura. Os classificadores serão treinados para identificar duas classes: benigna e maliciosa. A cada chamada do classificador, ele é retreinado e a partir daí ocorre a classificação. Por esse motivo, decidimos não incluir esse processo no modo detecção, pois ele é muito custoso.

#### 4.1.7 Detecção de maliciosidade

O processo de detecção é extremamente simples e rápido. Ele consiste em ler um *stream* de dados dos arquivos de traços e gerar como saída a classificação do traço como malicioso ou benigno. Os critérios para determinar a maliciosidade de uma amostra são os mesmos apresentados na Subseção 4.1.2. Esse processo, por padrão é finalizado na primeira ocorrência de maliciosidade. Ele foi adaptado para gerar um relatório das chamadas de sistema, que é utilizado no processo de obtenção de características dinâmicas.

### 4.2 MODO ANÁLISE

O funcionamento do sistema em modo análise é composto por uma união dos serviços apresentados nas Seções 4.1.1 e 4.1.2. Combinados, os dois serviços são utilizados para a geração de um vetor de características que representa a aplicação sob análise. Após a geração do vetor de características completo, ele é enviado para um processo de classificação que decide se a aplicação é benigna ou não. A Figura 4.2 apresenta a arquitetura do funcionamento do DroiDiagnosis.

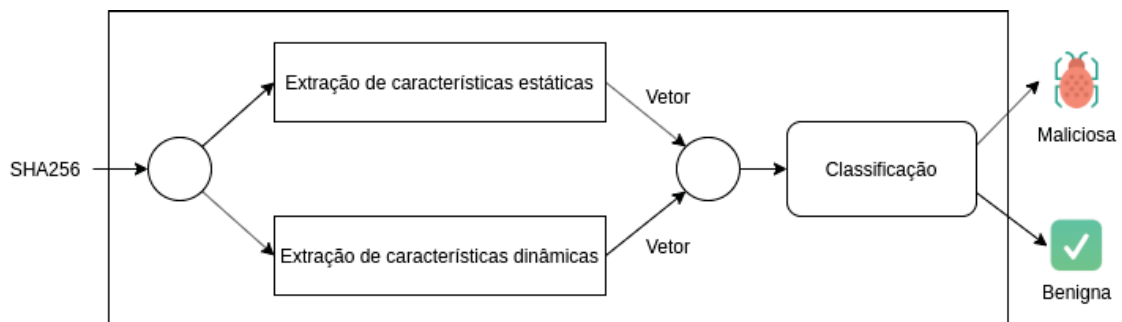


Figura 4.2: Modo de funcionamento de análise

Esse modo de operação é utilizado para os componentes de obtenção de características estáticas e dinâmicas, vetorização, repositório, classificação e uma adaptação do algoritmo de detecção de maliciosidade para obtenção de características dinâmicas.

### 4.3 MODO DIAGNÓSTICO

O modo diagnóstico utiliza apenas o componente de detecção de maliciosidades para operar. A função desse modo é ser utilizado para detecção de atividade maliciosa em traços pequenos de execução e não necessariamente no traço completo. Esse modo permite que *streams*

de traços sejam diagnosticados e também o diagnóstico de traços atemporais e independentes. A Figura 4.3 apresenta a comparação entre uma ferramenta de diagnóstico tradicional e a ferramenta para diagnóstico de traços mínimos.

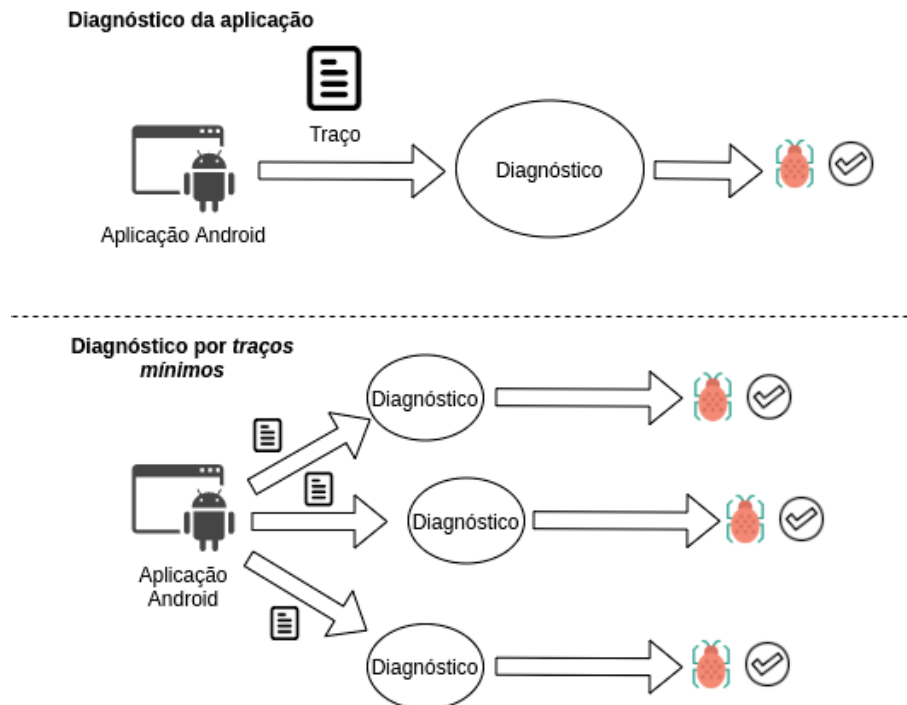


Figura 4.3: Comparação do diagnóstico tradicional e o diagnóstico de traços mínimos

Pela imagem, podemos perceber que a ideia é diagnosticar pedaços do traço da aplicação e não o traço completo, o que gera ganho de desempenho e possibilidade de paralelização. Através dessa arquitetura, esperamos que os aplicativos possam ser diagnosticados mais rapidamente e com menos custo utilizando as mesmas características. Em contrapartida, podemos prever que a análise do aplicativo não fará sentido nesse caso, pois ela não representa uma execução completa do mesmo.

## 5 RESULTADOS

Neste capítulo serão apresentados o experimento e resultados obtidos pelo DroiDiagnosis. Inicialmente apresentaremos a base de dados escolhida para os experimentos e a distribuição das classes. Explicaremos quais foram os experimentos realizados e as características extraídas das amostras durante a simulação, além das características extraídas de maneira estática. Mostraremos o desempenho do DroiDiagnosis no modo análise e no modo diagnóstico.

O objetivo do experimento é mostrar quais chamadas de sistemas podem ser usadas para diferenciar aplicações benignas de maliciosas, além de apresentar estatísticas que podem explicar alguns comportamentos comuns dessas classes. Além disso, mostraremos como o diagnóstico de traços mínimos pode ser uma alternativa ao diagnóstico de traço completo.

### 5.1 BASE DE DADOS

Neste trabalho a base de dados escolhida foi a Androzoo, (Allix et al., 2016). Esta base de dados é composta por 10.461.905 amostras de aplicativos do Android divididas entre benignas e maliciosas. Nem todas as aplicações possuem classificação. Para fins de testes, selecionamos 1.431 aplicativos benignos e 1.431 aplicativos maliciosos. Havíamos selecionado 2000 aplicações para cada classe, porém muitas aplicações maliciosas falharam a execução seja por incompatibilidade com a versão do Android, arquitetura utilizada ou por outro motivo não diagnosticado. Assim, para ter uma base balanceada, decidimos por igualar o número de aplicações por classe.

### 5.2 EXPERIMENTOS

Foi feito um único experimento com todas as aplicações mostradas na Seção 5.1. A máquina utilizada para as simulações possui uma CPU Intel(R) Core(TM) i5-7400 CPU @ 3.00GHz com 4 núcleos, 8 GB de memória RAM de 2.133 MHz. As aplicações foram executadas por um período de 20 segundos cada. Também foram coletados dados estáticos para análise de padrões. A versão do Android utilizada nas simulações foi a 7.0 (API 24) e arquitetura utiliza é a Intel X86\_64.

### 5.3 CHAMADAS DE SISTEMA

Os traços das chamadas de sistemas são linhas de texto em formato JSON que são obtidas através da conversão das saídas do Strace. O Código 5.1 mostra a saída for Strace formatada.

```

1 { "syscall": "faccessat",
2   "args": [ [ "AT_FDCWD" ], "/sbin/su",
3     { "name": "F_", "value": [ "OK" ] } ],
4   "result": "-1 EACCES (Permission denied)", "timing": 0.000026,
5   "pid": null, "type": "SYSCALL" }
```

Listing 5.1: Saída do Strace formatada para JSON

O campo **syscall** representa o nome da chamada de sistema; **args** são os argumentos da chamada (**/sbin/su**, que é o nome do arquivo que foi chamado); **result** representa o retorno da



chamada; **timing** o tempo de execução em *ms*; **pid** representa o *process id* da aplicação (como o *strace* está atrelado a um PID específico, ele retorna null); **type** representa o tipo de chamada.

Na Figura 5.1 apresentamos a quantidade média de chamadas de sistemas para ambas classes.

Verificamos que a quantidade de chamadas de *clock\_gettime*, *gettimeofday* e *sched\_yield* foram maiores para a classe dos *malware*. As funções *clock\_gettime* e *gettimeofday* são relativas a tempo, enquanto que função *sched\_yield* é utilizada para mover a **thread** para o fim da fila de execução da CPU. Um ponto que podemos destacar dos *goodware* é a chamada da função *madvise* que é utilizada para obter informações de memória junto ao *kernel* (não é controlado pelo processo). Acreditamos que esse valor tenha sido maior para os *goodware* pois eles possuem uma execução mais linear e estável que os *malware*, que em muitos casos apresentaram erros após um certo tempo de execução (muitos terminaram antes dos 20s).

Podemos perceber que os *malware* executaram mais chamadas de *readlinkat* e *exit* que os *goodware*. A função *readlinkat* tem a mesma função que *readlink*, e é utilizada para ler links simbólicos. A função *exit* é utilizada para finalizar um processo. Acreditamos que há uma relação entre essas duas chamadas, pois alguns *malware* costumam finalizar o processo quando há uma falha na leitura de um arquivo ou diretório.

Podemos verificar também que os *malware* efetuam mais chamadas de *umask*, *rt\_sigreturn*, *getrusage*, *getppid*, *getgid32*, *fnctl*, *exit\_group*, *execve* e *brk*. A função *umask* é utilizada para modificar a permissão de arquivos, *rt\_sigreturn* é o retorno do *signal handler*, *getrusage* é utilizada para obter a utilização de um recurso, *getppid* obtém o PID do processo, *getgid32* obtém a identidade de um grupo, *fnctl* manipula *file descriptors*, *exit\_group* finaliza todos as *threads* de um processo, *execve* executa um processo e *brk* altera a posição do *program break*. De maneira geral, a conclusão que chegamos é que os *malware* manipulam mais arquivos e buscam mais dados do processo que os *goodware*.

Na Figura 5.2 são apresentados os tempos médios gastos com chamadas de sistema por chamada de sistema. Como podemos observar, a quantidade de tempo gasto com chamadas de sistemas é proporcional a quantidade de vezes em que ela foi chamada.

Uma característica pouco explorada por outros trabalhos é ocorrência exclusiva de algumas chamadas de sistemas. A não ocorrência de chamadas de sistemas em uma das classes pode indicar que elas podem ser utilizadas para diferenciá-las. As chamadas de sistema exclusivas no conjunto de aplicações estudadas são:

- **rt\_sigsuspend**: espera por um sinal (apenas em *malware*)
- **chdir**: muda de diretório (apenas em *malware*)
- **pselect6**: monitora múltiplos *file descriptors* (apenas em *malware*)
- **mknodat**: cria um *node* em um diretório (apenas em *goodware*)
- **socket**: cria um ponto de comunicação (apenas em *goodware*)
- **fsync**: sincroniza dados em *buffers* com o dispositivo de memória permanente (apenas em *goodware*)

## 5.4 DIRETÓRIOS E ARQUIVOS

Para diagnóstico de diretórios e arquivos acessados pelas aplicações, utilizamos os parâmetros de algumas chamadas de sistemas. Os arquivos e pastas são obtidos através da análise

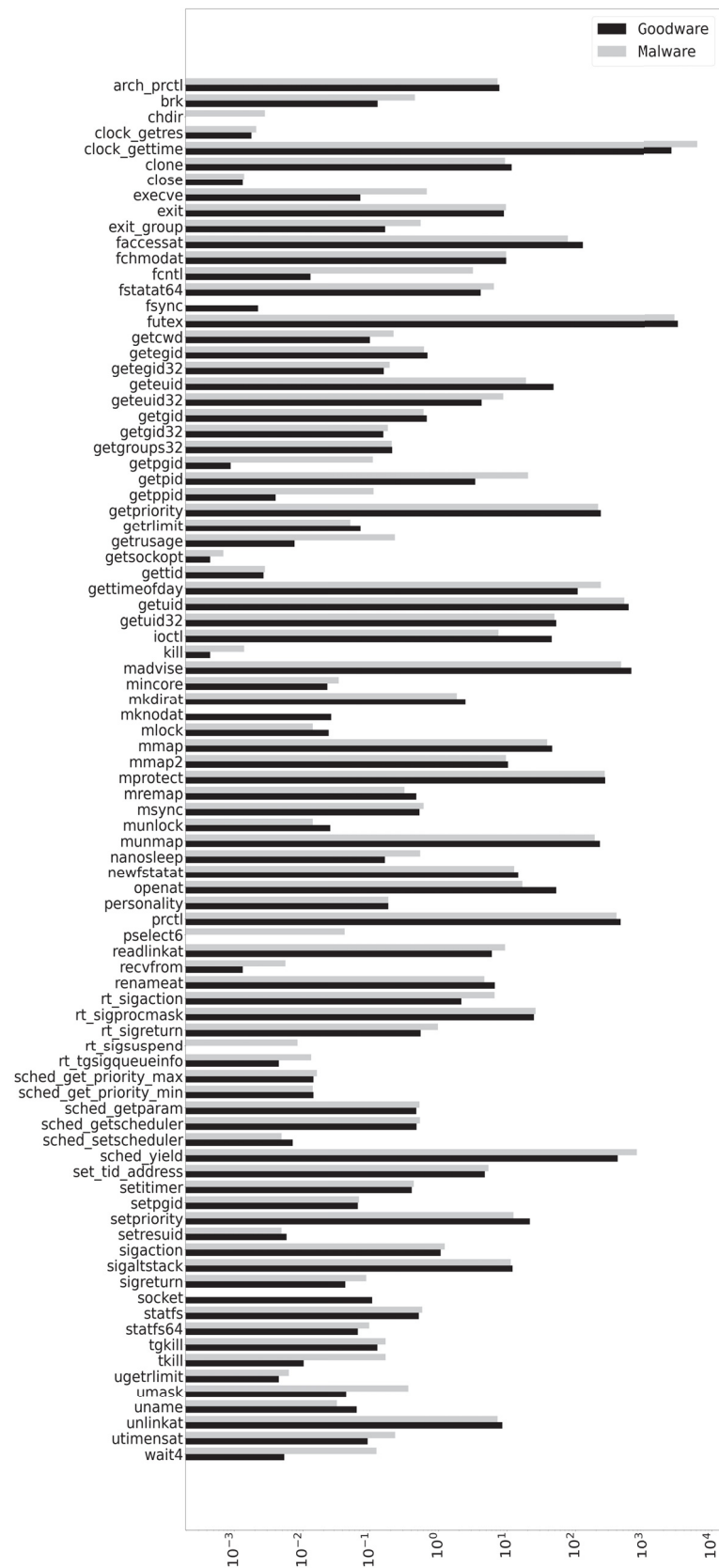


Figura 5.1: Quantidade média de chamadas de sistema por classe de aplicação

de parâmetros das chamadas **unlinkat**, **fchmodat**, **faccessat** e **newfstatat**. Para facilitar a análise, dividimos os arquivos/diretórios de acordo com o profundidade  $n$ , sendo que o  $n$  é definido



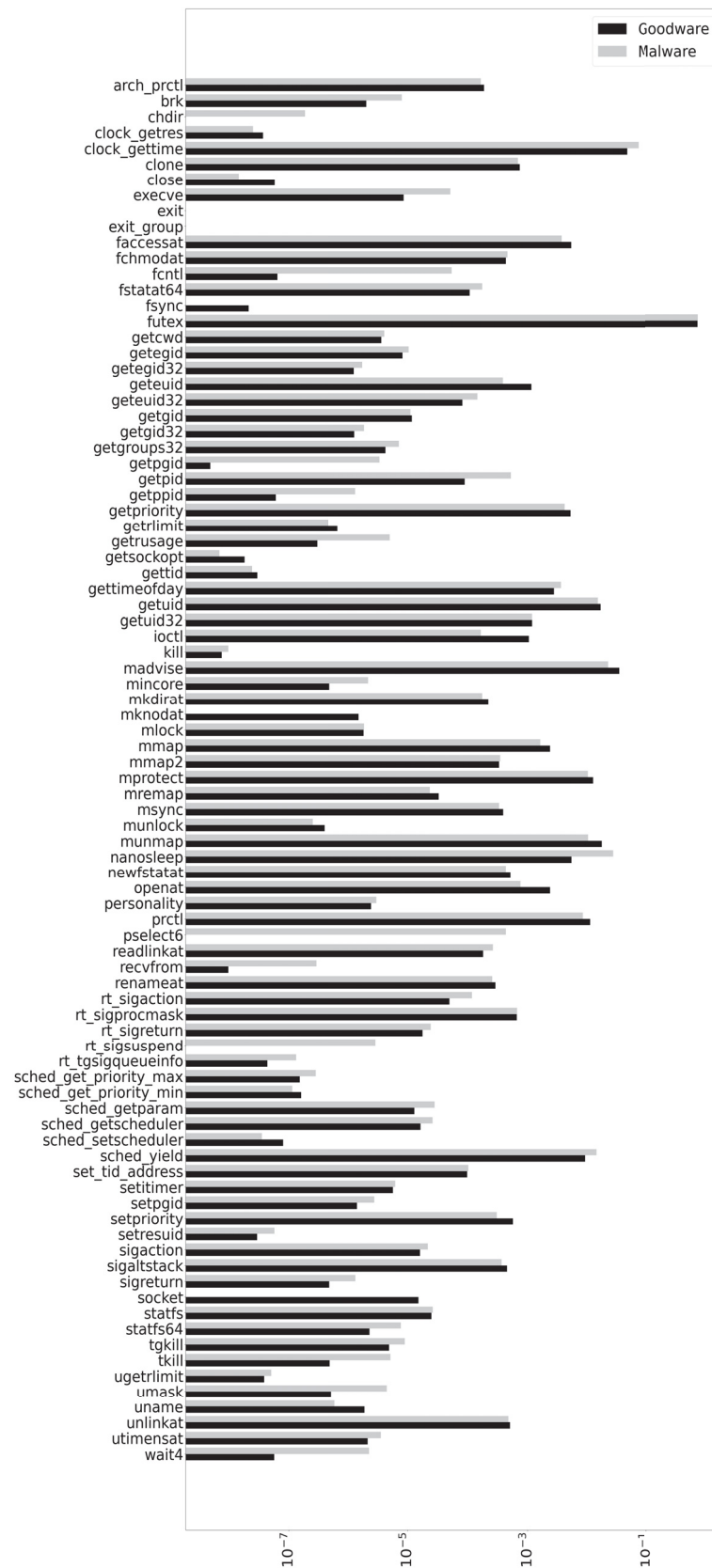


Figura 5.2: Tempo médio (em ms) de execução de chamadas de sistema por classe de aplicação

como a **quantidade de diretórios entre a pasta principal do sistema e o arquivo/diretório acessado**.

Neste trabalho focamos nos arquivos/diretórios de profundidade 0, 1 e 2. Como as aplicações acessam muitos arquivos, optamos por apresentar apenas os 10 diretórios mais acessados por cada classe de aplicação em cada profundidade. A Figura 5.3 apresenta os diretórios de profundidade 0 mais acessados por *malware* e *goodware* (diretórios que não tiveram nenhum acesso em alguma das classes também foram incluídos).

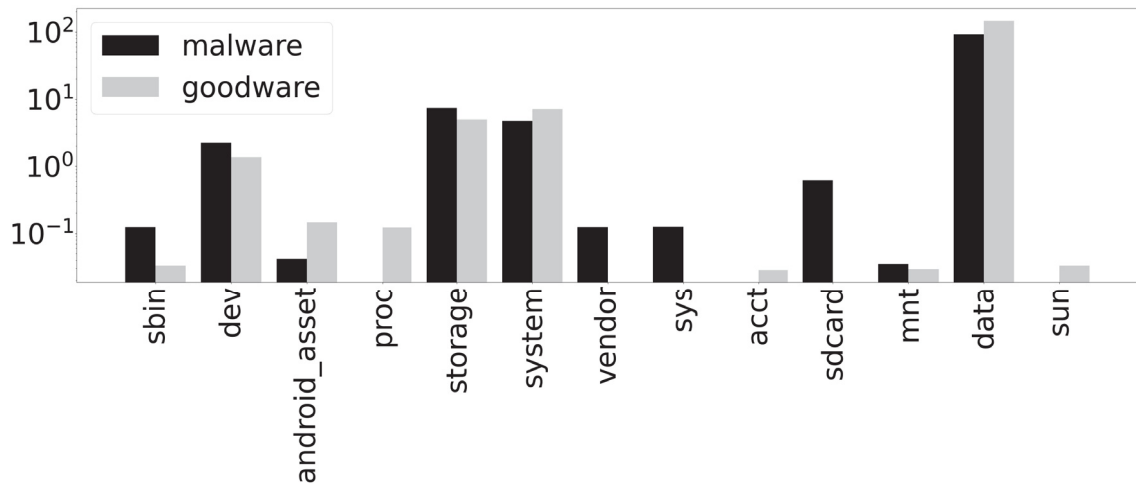


Figura 5.3: Quantidade média de acessos por diretório de profundidade 0

Podemos observar que a maioria dos diretórios acessados pelos *goodware* e pelos *malware* são os mesmos. No entanto, diretórios como **sdcard**, **sys** e **vendor** são mais acessados por *malware* do que por *goodware*. O diretório *sdcard* é mapeado para o cartão de memória do dispositivo, enquanto que os diretórios **sys** e **vendor** têm permissão permitida apenas para o usuário **root** do sistema.

A Figura 5.4 apresenta os 10 diretórios de profundidade 1 mais acessados por *malware* e *goodware* (diretórios que não tiveram nenhum acesso em alguma das classes também foram incluídos).

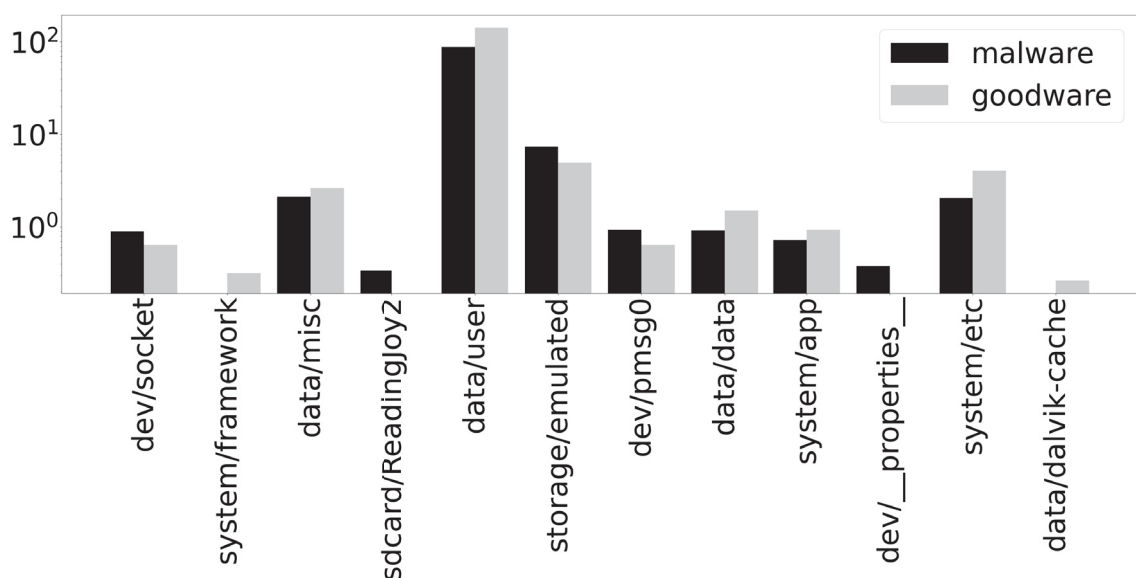


Figura 5.4: Quantidade média de acessos por diretório de profundidade 1

No gráfico, não notamos muitas diferenças entre os diretórios mais acessados por *goodware* e *malware*, porém podemos notar que a média de acessos ao diretório **storage/emulated**

é maior para os *malware* do que para os *goodware*. Esse diretório representa a memória interna do dispositivo, o que indica que em média, aplicações maliciosas acessam mais a memória interna do que aplicações benignas.

A Figura 5.5 apresenta os 10 diretórios de profundidade 2 mais acessados por *malware* e *goodware*.

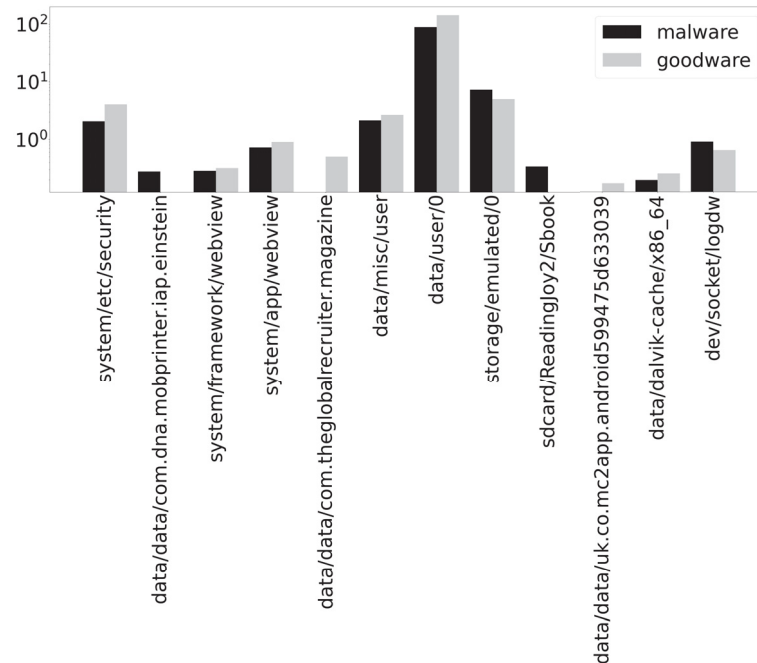


Figura 5.5: Quantidade média de acessos por diretório de profundidade 2

No gráfico, não notamos muitas diferenças entre os diretórios mais acessados por *goodware* e *malware*, porém verificamos uma semelhança na quantidade média de acessos a alguns diretórios, como por exemplo **data/user/0**, **storage/emulated/0**, **data/misc/user** e **system/etc/security**. A média de acessos a esses diretórios pode indicar um padrão de acesso de aplicações no Android. Além disso, assim como nos diretórios de profundidade 1, podemos notar que, em diretórios de profundidade 2, novamente os *malware* acessam mais a memória interna do que os *goodware*.

Algumas aplicações tentaram realizar trocas de usuário através do comando **su**. A Figura 5.6 mostra que tanto aplicativos maliciosos quanto benignos tentaram fazer troca de usuário, sendo que nas amostras analisadas aplicativos benignos tentaram trocar de usuário mais vezes que aplicativos maliciosos.

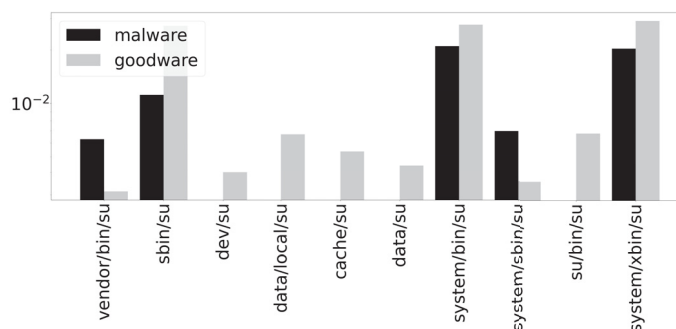


Figura 5.6: Quantidade média de tentativas de troca de usuário

## 5.5 DADOS DE REDE

Nesta seção apresentaremos os 10 domínios acessados por aplicações maliciosas e não-maliciosas. A Figura 5.7 mostra os domínios mais acessados pelos *malware* e pelos *goodware*.

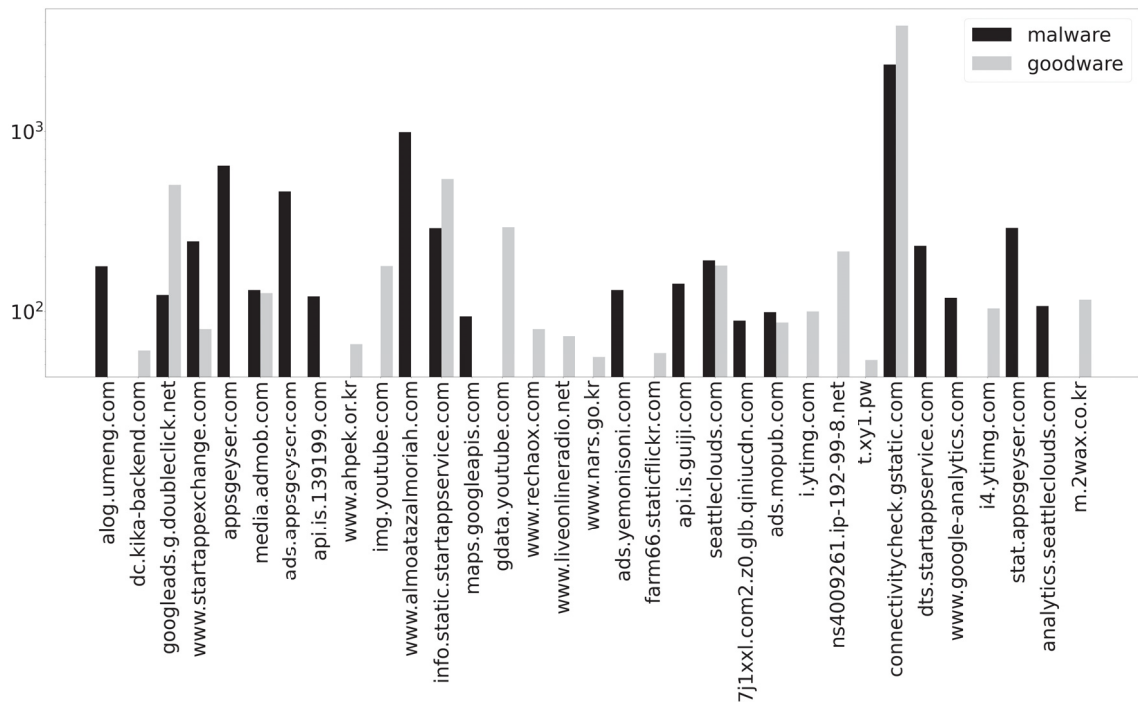


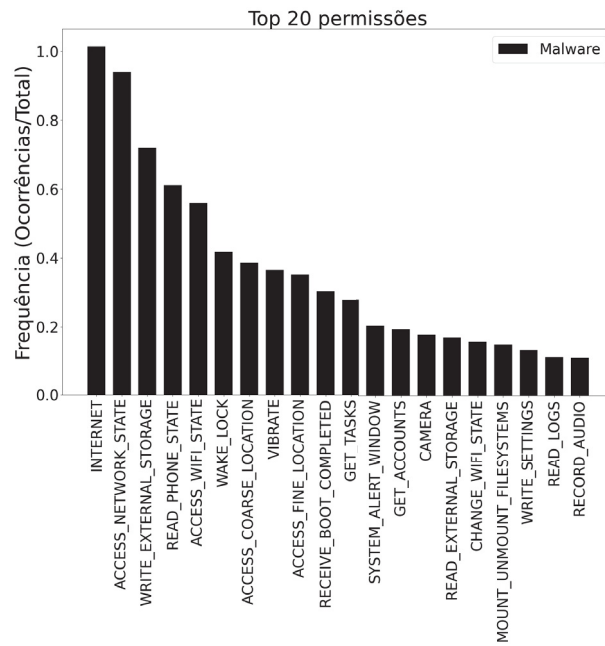
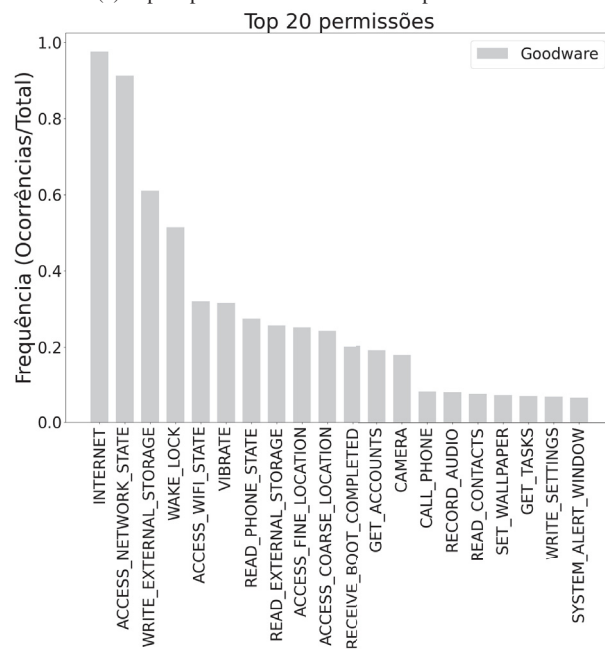
Figura 5.7: Quantidade média de acessos por domínio

Verificando o gráfico podemos notar algumas características, como por exemplo o fato de o domínio *connectivitycheck.gstatic.com* ser o mais acessado tanto por *goodware* quanto *malware*, indicando que pode ser um padrão do Android. Outra característica é a presença de domínios associados à *adware* e propaganda, sendo que eles aparecem tanto em aplicações maliciosas como não maliciosas. Geralmente domínios associados a propaganda são os iniciados por *ad.*, *ads.* ou *googleads.*, mas existem exceções, como por exemplo *media.admob.com*.

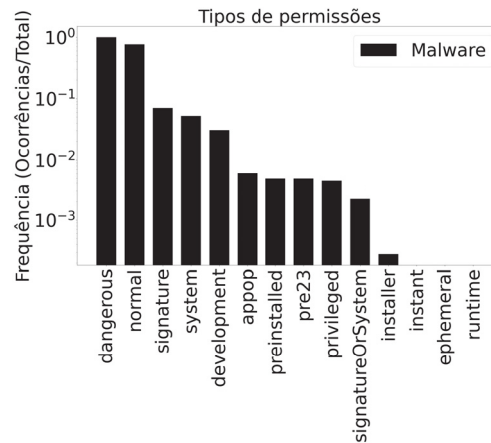
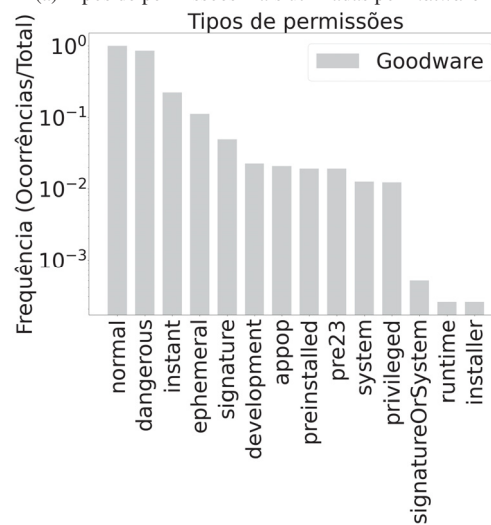
## 5.6 PERMISSÕES

As permissões são um dos pontos fundamentais da segurança do Android e um limitador de acesso a recursos do dispositivo e dados do usuário pelas aplicações. As permissões normais são liberadas automaticamente pelo Android, pois são pouco ameaçadoras a segurança do usuário, as assinadas são usadas apenas por aplicativos do sistema e as perigosas podem acessar dados sensíveis e comprometer funcionalidades do dispositivo. Nesta seção iremos apresentar os padrões de permissões utilizados por aplicações benignas e maliciosas através da análise estática.

As Figuras 5.8(a) e 5.8(b) apresentam as 20 permissões mais utilizadas por *malware* e *goodware*. Ambas classes de aplicações requisitam muitas permissões de acesso à rede, dados do dispositivo e memória externa. Algumas permissões requisitadas por *malware* podem ser destacadas, como: **GET\_TASKS**, que garante acesso ao controle de *tasks* do Android, **CHANGE\_WIFI\_STATE**, que permite alterar o estado da conexão *Wifi*, **MOUNT\_UNMOUNT\_FILESYSTEMS**, que permite montar/desmontar arquivos de sistema para armazenamento removível e **READ\_LOGS**, que permite ler *logs* do sistema.

(a) Top 20 permissões mais utilizadas por *malware*(b) Top 20 permissões mais utilizadas por *goodware*

As Figuras 5.9(a) e 5.9(b) apresentam os tipos de permissões que mais foram requisitadas por aplicações maliciosas e não-maliciosas. É possível observar que as aplicações benignas requisitam tantas permissões perigosas quanto as aplicações maliciosas, no entanto a quantidade de permissões normais é maior nas aplicações benignas que nas maliciosas. Outro ponto é que permissões assinadas (*signature*) dificilmente são utilizadas por qualquer uma das classes.

(a) Tipos de permissões mais utilizadas por *malware*(b) Tipos de permissões mais utilizadas por *goodware*

## 5.7 ANÁLISE

Nesta seção apresentaremos alguns resultados relacionados ao aprendizado de máquina, onde utilizaremos os classificadores SVM, KNN e Decision Tree para classificar as aplicações em maliciosas e não maliciosas. Como características para construção do modelo, iremos utilizar o conjunto de resultados obtidos neste capítulo.

### 5.7.1 Métricas

Para que o desempenho do nosso modelo possa ser avaliado através dos classificadores, devemos utilizar as métricas: matriz de confusão, acurácia, precisão, *recall* e *f1-score*.

#### 5.7.1.1 Matriz de confusão

A matriz de confusão é uma tabela que combina as classes de um problema de classificação e é utilizada para verificar o desempenho de classificadores. Os valores da tabela são as quantidades de VP (Verdadeiros-positivos), FN (Falsos-negativos), FP (Falsos-positivos) e VN (Verdadeiros-negativos). A Figura 5.10 apresenta um exemplo de matriz de confusão. Para o problema em questão, os valores da matriz de confusão para as classes *goodware* e *malware* são:

- *VP*: quantidade de aplicações classificadas como *goodware* que realmente são *goodware*

- *FN*: quantidade de aplicações classificadas como *malware* que na verdade são *goodware*
- *FP*: quantidade de aplicações classificadas como *goodware* que na verdade são *malware*
- *VN*: quantidade de aplicações classificadas como *malware* que realmente são *malware*

		PREDITO	
		Classe A	Classe B
VERDADEIRO	Classe A	VP	FN
	Classe B	FP	VN

Figura 5.10: Matriz de confusão

#### 5.7.1.2 Acurácia

A acurácia é basicamente o número de acertos (positivos) dividido pelo total de exemplos, ou entre outras palavras, o quão frequente o classificado está correto. A formula da acurácia pode ser definida por

$$Acuracia = (VP + VN)/n$$

onde  $n$  é o número total de amostras da base de dados.

#### 5.7.1.3 Precisão

A precisão é uma métrica que indica quantas amostras de uma determinada classe que são realmente daquela classe (VP), dividido pela soma entre VP e o número de exemplos classificados nesta classe, mas que pertencem a outra (FP). Em outras palavras: daqueles que classifiquei como corretos, quantos efetivamente eram? A formula da precisão pode ser definida por

$$Precisao = VP/(VP + FP)$$

#### 5.7.1.4 Recall

O *recall* é uma métrica que tenta responder a seguinte questão: qual a proporção de positivos foi identificado corretamente? A formula do *recall* pode ser definido por

$$Recall = VP/(VP + FN)$$

#### 5.7.1.5 F1-score

A métrica *F1-score* é uma combinação das métricas precisão e *recall* a fim de encontrar um balanceamento entre elas. A formula do *F1-score* pode ser definida por

$$f1 - score = 2 * ((Precisao * Recall)/(Precisao + Recall))$$

### 5.7.2 Características

As 285 características utilizadas no modelo são divididas em dinâmicas e estáticas. Foram utilizadas 242 características dinâmicas, distribuídas entre quantidade média de chamadas de sistema, tempo médio de chamadas de sistema, domínios acessados, e pastas acessadas. Também foram selecionadas 43 características estáticas, distribuídas entre permissões e tipos de permissões.

### 5.7.3 Base de dados

As amostras foram selecionadas da base do Androzoo, as mesmas utilizadas nas simulações. No entanto, como o número de aplicações benignas é maior que o número de aplicações maliciosas, optamos por balancear a base e igualar o número de aplicações de cada classe. O número de aplicações benignas e maliciosas utilizadas para os testes é de 2.862 divididas entre as duas classes. Além disso, dividimos as amostras em 70% base de treinamento e 30% base de testes, dessa forma temos 1.003 amostras para treinamento do modelo e 428 amostras para teste.

### 5.7.4 Análise dinâmica

Com as características dinâmicas apresentadas, testamos a eficiência dos três classificadores utilizando somente essas características. Com o classificador SVM obtivemos a Matriz de Confusão 5.11. Da tabela podemos verificar que das 428 amostras maliciosas, 346 foram classificadas corretamente como maliciosas. Das 428 aplicações benignas, 254 foram classificadas como benignas.

		Valores preditos		Total
		Goodware	Malware	
Valores reais	Goodware	254	174	428
	Malware	82	346	428
Total		336	520	856

Figura 5.11: Matriz de confusão para a análise dinâmica utilizando SVM

Utilizando o classificador KNN obtivemos a Matriz de Confusão 5.12, onde podemos verificar que das 428 amostras maliciosas, 303 foram classificadas corretamente como maliciosas e, das 428 aplicações benignas, 282 foram classificadas como benignas.

		Valores preditos		Total
		Goodware	Malware	
Valores reais	Goodware	282	146	428
	Malware	125	303	428
Total		407	449	856

Figura 5.12: Matriz de confusão para a análise dinâmica utilizando KNN

Utilizando o classificador Decision Tree obtivemos a Matriz de Confusão 5.13. Da tabela podemos verificar que das 428 amostras maliciosas, 295 foram classificadas corretamente como maliciosas e, das 428 aplicações benignas, 260 foram classificadas como benignas.

Baseados nas matrizes de confusão obtidas, podemos definir as métricas que qualificam este modelo pela Tabela 5.1. Na tabela podemos verificar que o SVM foi o classificador que



		Valores preditos		Total
		Goodware	Malware	
Valores reais	Goodware	260	168	428
	Malware	133	295	428
Total		393	463	856

Figura 5.13: Matriz de confusão para a análise dinâmica utilizando Decion-Tree

obteve o melhor desempenho (71% de precisão), sendo que cerca de 80% (346/428) das amostras maliciosas foram classificadas corretamente.

Classificador	Acurácia	Precisão	Recall	F1-Score
SVM	<b>0.701</b>	<b>0.711</b>	<b>0.701</b>	<b>0.697</b>
KNN	0.683	0.684	0.683	0.683
Decision Tree	0.648	0.649	0.648	0.648

Tabela 5.1: Métricas de desempenho da análise dinâmica

#### 5.7.5 Análise estática

Com as características dinâmicas apresentadas, testamos a eficiência dos três classificadores utilizando somente essas característica. O classificador SVM obteve a Matriz de Confusão 5.14, onde podemos verificar que das 428 amostras maliciosas, 357 foram classificadas corretamente como maliciosas, e das 428 aplicações benignas, 275 foram classificadas como benignas.

		Valores preditos		Total
		Goodware	Malware	
Valores reais	Goodware	275	153	428
	Malware	71	357	428
Total		346	510	856

Figura 5.14: Matriz de confusão para a análise estática utilizando SVM

Utilizando o classificador KNN obtivemos a Matriz de Confusão 5.15. Na tabela podemos verificar que das 428 amostras maliciosas, 306 foram classificadas corretamente como maliciosas, das 428 aplicações benignas, 278 foram classificadas como benignas.

		Valores preditos		Total
		Goodware	Malware	
Valores reais	Goodware	278	150	428
	Malware	122	306	428
Total		400	456	856

Figura 5.15: Matriz de confusão para a análise estática utilizando KNN

Com o classificador Decision Tree obtivemos a Matriz de Confusão 5.16, onde podemos verificar que das 428 amostras maliciosas, 284 foram classificadas corretamente como maliciosas, e das 428 aplicações benignas, 283 foram classificadas como benignas.

		Valores preditos		Total
		Goodware	Malware	
Valores reais	Goodware	275	153	428
	Malware	71	357	428
Total		346	510	856

Baseados nas matrizes de confusão obtidos, podemos definir as métricas que qualificam este modelo pela Tabela 5.2. Da tabela podemos verificar que o SVM foi o classificador que obteve o melhor desempenho, com precisão de 74.7%.

Classificador	Acurácia	Precisão	Recall	F1-Score
SVM	<b>0.738</b>	<b>0.747</b>	<b>0.738</b>	<b>0.736</b>
KNN	0.682	0.683	0.682	0.682
Decision Tree	0.662	0.662	0.662	0.662

Tabela 5.2: Métricas de desempenho da análise estática

### 5.7.6 Análise híbrida

Utilizando as características dinâmicas e estáticas e os classificadores escolhidos obtivemos as métricas apresentadas na Sub-Seção 5.7.1 para medir o desempenho do modelo. O classificador SVM obteve a Matriz de Confusão 5.17, onde podemos verificar que das 428 amostras maliciosas, 347 foram classificadas corretamente como maliciosas, e das 428 aplicações benignas, 338 foram classificadas corretamente.

		Valores preditos		Total
		Goodware	Malware	
Valores reais	Goodware	338	90	428
	Malware	81	347	428
Total		419	437	856

Figura 5.17: Matriz de confusão para a análise híbrida utilizando SVM

Utilizando o classificador KNN obtivemos a Matriz de Confusão 5.18, onde 428 amostras maliciosas, 342 foram classificadas corretamente como maliciosas, e das 428 aplicações benignas, 311 foram classificadas como benignas.

		Valores preditos		Total
		Goodware	Malware	
Valores reais	Goodware	338	90	428
	Malware	81	347	428
Total		419	437	856

Figura 5.18: Matriz de confusão para a análise híbrida utilizando KNN

Utilizando o classificador Decision Tree obtivemos a Matriz de Confusão 5.19 onde das 428 amostras maliciosas, 293 foram classificadas corretamente, e das 428 aplicações benignas, 283 foram classificadas como benignas.

		Valores preditos		Total
		Goodware	Malware	
Valores reais	Goodware	283	145	428
	Malware	135	293	428
Total		418	438	856

Figura 5.19: Matriz de confusão para a análise híbrida utilizando Decision-Tree

Baseado nas matrizes de confusão obtidos, podemos definir as métricas que qualificam este modelo pela Tabela 5.3. Da tabela podemos verificar que o SVM foi o classificador que obteve o melhor desempenho, com 80% de precisão.

<b>Classificador</b>	<b>Acurácia</b>	<b>Precisão</b>	<b>Recall</b>	<b>F1-Score</b>
SVM	<b>0.800</b>	<b>0.800</b>	<b>0.800</b>	<b>0.800</b>
KNN	0.763	0.764	0.763	0.763
Decision Tree	0.673	0.673	0.673	0.673

Tabela 5.3: Métricas de desempenho da análise híbrida

#### 5.7.7 Comparação entre os modelos

Comparando os resultados das análises estática, dinâmica e híbrida, podemos perceber que em todas elas o classificador SVM obteve os melhores resultados, seguido do KNN e do Decision-Tree. Outro ponto de destaque é que a análise estática obteve melhores resultados que a análise dinâmica. Acreditamos que esse resultado é devido ao fato de as características estáticas extraídas no experimentos destacarem mais as aplicações maliciosas das benignas do que as dinâmicas.

### 5.8 TRAÇOS MÍNIMOS

Nesta seção mostraremos os principais resultados referentes ao diagnóstico de aplicações utilizando traços mínimos. Analisaremos as quantidades de chamadas de sistemas em alguns momentos de uma aplicação maliciosas, além de verificar quantos fragmentos são necessários para identificar um fragmento malicioso.

#### 5.8.1 Quantidade de fragmentos maliciosos

Para identificar a quantidade de fragmentos de traço necessários para identificar um traço parcial malicioso, selecionamos 10 aplicações maliciosas aleatórias e as dividimos em 40, 100, 200 e 400 fragmentos, e as reprocessamos utilizando os mecanismos de diagnóstico do DroiDiagnosis. A Figura 5.20 apresenta a quantidade de fragmentos maliciosos pela quantidade de fragmentos.

Podemos notar que conforme aumenta o número de fragmentos, maior o número de fragmentos maliciosos identificados. Porém, isso não indica que a probabilidade de encontrar um fragmento malicioso será grande. Por outro lado, uma quantidade maior de fragmentos permite que todos sejam processados em paralelo e o diagnóstico seja encontrado mais rápido do que processar o traço completo.

#### 5.8.2 Quantidade de chamadas de sistemas

Para verificar as quantidade de chamadas de sistemas por fragmentos, selecionamos 10 aplicativos maliciosos e 10 benignos, juntamos seus traços em dois arquivos com traços maliciosos e benignos, e os dividimos em 2, 5, 10 e 20 partes. Os resultados mostraram que tanto aplicativos maliciosos quanto benignos tendem a manter a quantidade de chamadas de sistemas proporcionais e pouco variável com o aumento das divisões dos traços, como mostrado nas Figuras 5.21 e 5.22.

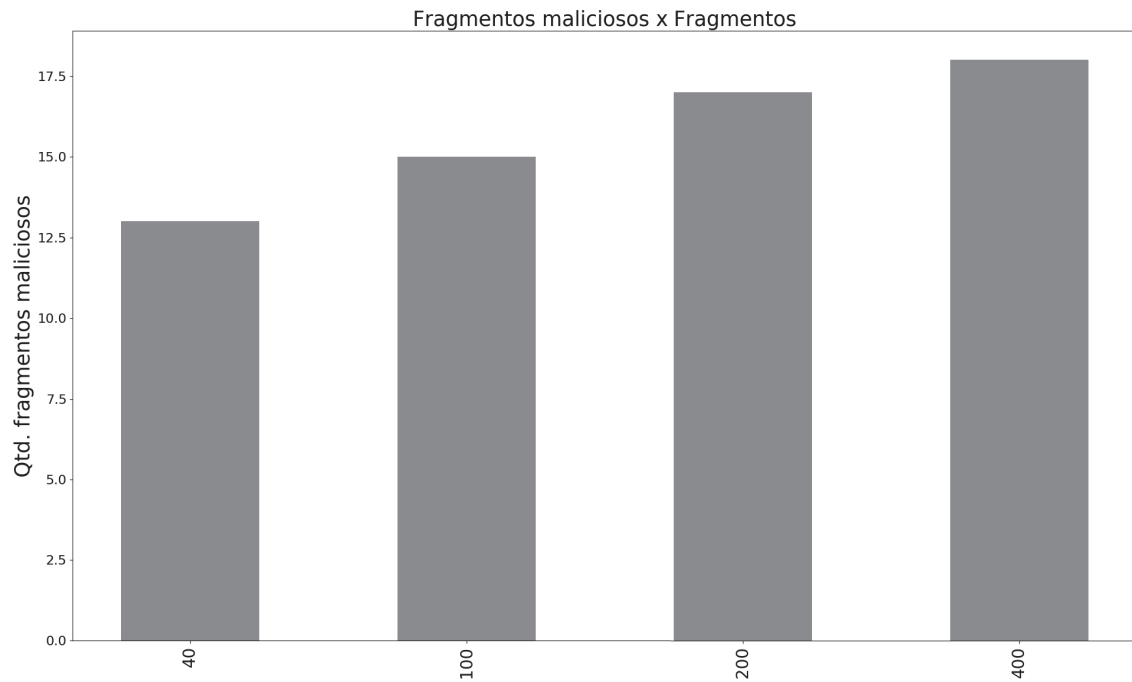


Figura 5.20: Fragmentos maliciosos encontrados pela quantidade fragmentos

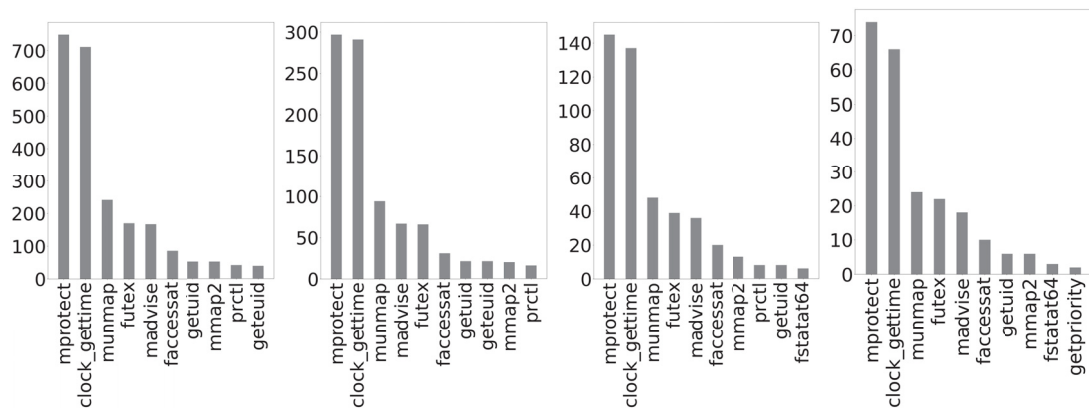


Figura 5.21: Top 10 chamadas de sistema em 2, 5, 10 e 20 divisões de traços maliciosos

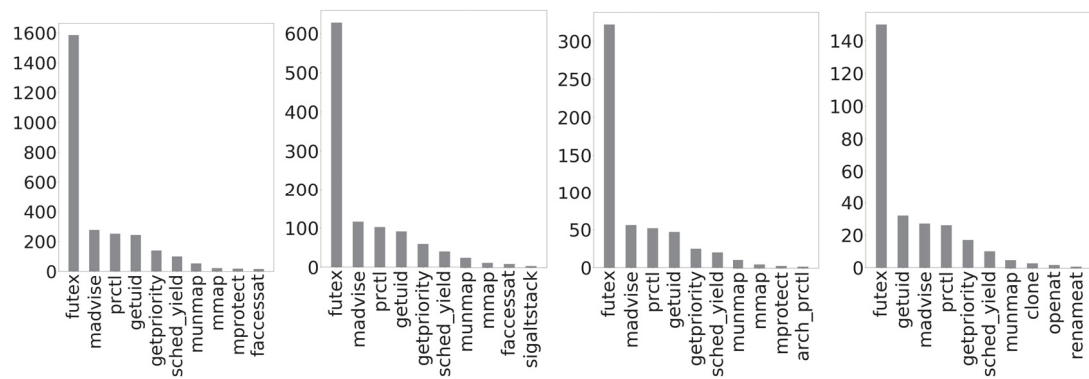


Figura 5.22: Top 10 chamadas de sistema em 2, 5, 10 e 20 divisões de traços benignos

## 5.9 CONSIDERAÇÕES FINAIS

Como podemos ver pelos resultados, a análise híbrida apresentou melhor desempenho em relação às análises estática e dinâmica, sendo em que todos os modelos, o classificador SVM obteve os melhores resultados. Apesar disso, na literatura encontramos soluções que possuem taxas de precisão melhores que as apresentadas neste trabalho. No entanto, cremos que com uma base de dados e um tempo de execução maiores, a classificação utilizando chamadas de sistema pode ser mais precisa. Quanto ao diagnóstico utilizando traços mínimos, concluímos que quanto maior o número de fragmentos maior o número de fragmentos maliciosos, o que indica que o fragmento do traço de um aplicativo pode ser usado como alternativa ao traço completo.

### 5.9.1 Aplicabilidade

Para diagnosticar uma aplicação utilizando o DroidDiagnosis, seria necessário um agente instalado no dispositivo para enviar os traços de uma ou mais aplicações de tempos em tempos. Nesse caso, cada traço seria processado de maneira independente dos traços anteriores. O DroidDiagnosis também pode ser usado para análise de aplicações em massa, bastando apenas que fossem definidas uma base de treinamento e uma base de teste.

## 6 CONCLUSÃO E TRABALHOS FUTUROS

Neste trabalho apresentamos o DroiDiagnosis, uma ferramenta baseada no rastreamento de chamadas de sistemas para identificar e diagnosticar aplicações maliciosas no Android. Pelo fato de o Android ser baseado no Linux, utilizamos a ferramenta **strace** como base para nossa implementação.

Ao contrário de outros trabalhos, não focamos apenas na quantidade de chamadas de sistema efetuadas pelas aplicações, pois acreditamos que essa métrica não garante que a maliciosidade de uma aplicação. No entanto, verificamos, através dos experimentos, que algumas chamadas ocorrem em média mais vezes nos *malware* do que nos *goodware*. Além disso, verificamos que algumas chamadas de sistema ocorreram em uma classe e na outra não.

Além das chamadas de sistema, também verificamos as URLs acessadas pelas aplicações assim como os diretórios. Através da análise dos diretórios, identificamos que as aplicações maliciosas e não maliciosas tendem a acessar em média os mesmos diretórios, mas os *malware* acessam diretórios com acesso restrito mais vezes que os *goodware*. Identificamos também alguns diretórios que são acessados de maneira padrão por aplicações do Android, como *data*, *system*, *storage* e *dev*. Detectamos também várias tentativas de utilização do comando **su** por aplicações maliciosas. Com relação às URLs acessadas, identificamos que tanto os *malware* quanto os *goodware* mostram propagandas através de uma análise manual dos domínios acessados.

Em adição à análise das chamadas de sistemas, realizamos engenharia-reversa no código das aplicações para verificar algumas outras características relevantes, em especial as permissões declaradas no manifesto. Verificamos que tanto aplicações maliciosas quanto benignas tendem a utilizar permissões perigosas, no entanto aplicações benignas utilizam mais permissões ditas normais do que as maliciosas. Além disso, verificamos que algumas permissões, como **GET\_TASKS**, **CHANGE\_WIFI\_STATE**, **MOUNT\_UNMOUNT\_FILESYSTEMS** e **READ\_LOGS** costumam ser mais utilizadas pelos *malware*.

Utilizamos os dados obtidos nos experimentos para identificação de padrões de *malware* e *goodware* e classificação através de aprendizado de máquina. Criamos três modelos baseados em características de análise dinâmica, estática e híbrida, uma combinação das duas. Utilizamos os classificadores SVM, KNN e *Decision Tree*, sendo que o SVM foi melhor em todos os testes. Verificamos também que o modelo estático se saiu melhor que o modelo dinâmico, sendo que o estático obteve o máximo de 73% de amostras classificadas corretamente, enquanto que o dinâmico obteve 70%. Melhor do que os dois, o modelo híbrido classificou 80% das amostras corretamente.

Por fim, utilizamos o DroiDiagnosis para detecção de fragmentos de traços maliciosos, e concluímos que o aumento do número de fragmentos de um traço diminui a probabilidade de um fragmento ser malicioso, mas permite que mais fragmentos sejam processados em paralelo, o que possibilita que mais aplicações sejam diagnosticadas ao mesmo tempo.

### 6.1 PROBLEMAS CONHECIDOS

Algumas vulnerabilidades encontradas durante o desenvolvimento deste trabalho:

- Algumas aplicações podem executar atividade quando ocorre o *boot* do sistema. Nesse caso, nossa solução não detecta essa atividade maliciosa em específico pois assumimos que o *boot* já foi realizado

- Como citado por (Denney et al., 2018) alguns tipos de *rootkits* não utilizam chamadas de sistemas.
- Por utilizar o simulador, o DroiDiagnosis pode não identificar *malware* que detecta que está sendo executado em ambiente simulado e não efetua atividades maliciosas
- Ao utilizar a ferramenta Monkey para simular o usuário, algumas funcionalidades da aplicação podem não ser exploradas pois não foram acionadas

Em trabalhos futuros pretendemos contornar os problemas identificados e efetuar testes com bases de dados maiores que a utilizada.

## REFERÊNCIAS

- Abel, R. (2018). Social media and engineering used to spread tempted cedar spyware. <https://www.scmagazine.com/tempted-cedar-spyware-spread-in-fake-kik-messenger-app/article/746148/>. Acessado em 20/08/2018.
- ADB (2019). Android debug bridge. <https://developer.android.com/studio/command-line/adb?hl=pt-br>. Acessado em 05/01/2019.
- Ahsan-Ul-Haque, A. S. M., Hossain, M. S. e Atiquzzaman, M. (2018). Sequencing system calls for effective malware detection in android. Em *2018 IEEE Global Communications Conference (GLOBECOM)*, páginas 1–7.
- Alazab, M., Moonsamy, V., Batten, L., Lantz, P. e Tian, R. (2012). Analysis of malicious and benign android applications. Em *2012 32nd International Conference on Distributed Computing Systems Workshops*, páginas 608–616.
- Allix, K., Bissyandé, T. F., Klein, J. e Traon, Y. L. (2016). Androzoo: Collecting millions of android apps for the research community. Em *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, páginas 468–471.
- AndroidSourceProject (2016). Androidsource. <https://source.android.com/>. Acessado em 06/09/2018.
- AndroidStudio (2019). Run apps on the android emulator. <https://developer.android.com/studio/run/emulator>. Acessado em 06/11/2019.
- Anthony Desnos, Geoffroy Gueguen, S. B. (2018). Androguard. <https://androguard.readthedocs.io/en/latest/>. Acessado em 10/01/2020.
- Arp, D., Spreitzenbarth, M., Hübner, M., Gascon, H. e Rieck, K. (2014). Drebin: Effective and explainable detection of android malware in your pocket. Em *Symposium on Network and Distributed System Security (NDSS)*.
- Arshad, S., Shah, M. A., Wahid, A., Mehmood, A., Song, H. e Yu, H. (2018). Samadroid: A novel 3-level hybrid malware detection model for android operating system. *IEEE Access*, 6:4321–4339.
- Asmitha, K. A. e Vinod, P. (2014). A machine learning approach for linux malware detection. Em *2014 International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT)*, páginas 825–830.
- Bermejo, L. (2017). Android backdoor ghostctrl can silently record your audio, video, and more. <https://blog.trendmicro.com/trendlabs-security-intelligence/android-backdoor-ghostctrl-can-silently-record-your-audio-video-and-more/>. Acessado em 19/08/2018.
- Bhatia, T. e Kaushal, R. (2017). Malware detection in android based on dynamic analysis. Em *2017 International Conference on Cyber Security And Protection Of Digital Services (Cyber Security)*, páginas 1–6.



- Bläsing, T., Batyuk, L., Schmidt, A., Camtepe, S. A. e Albayrak, S. (2010). An android application sandbox system for suspicious software detection. Em *2010 5th International Conference on Malicious and Unwanted Software*, páginas 55–62.
- Bocek, V. e Chrysaidos, N. (2018). Avast threat labs analyzed malware that has affected thousands of users around the world. <https://blog.avast.com/android-devices-ship-with-pre-installed-malware>. Acessado em 20/08/2018.
- Buchka, N. (2017). Skygofree: Following in the footsteps of hackingteam. <https://securelist.com/skygofree-following-in-the-footsteps-of-hackingteam/83603/>. Acessado em 20/08/2018.
- Burguera, I., Zurutuza, U. e Nadjm-Tehrani, S. (2011). Crowdroid: Behavior-based malware detection system for android. páginas 15–26.
- Dash, S. K., Suarez-Tangil, G., Khan, S., Tam, K., Ahmadi, M., Kinder, J. e Cavallaro, L. (2016). Droidscribe: Classifying android malware based on runtime behavior. Em *2016 IEEE Security and Privacy Workshops (SPW)*, páginas 252–261.
- Denney, K., Kaygusuz, C. e Zuluaga, J. (2018). A survey of malware detection using system call tracing techniques.
- Dent, S. (2018). Report finds android malware pre-installed on hundreds of phones. <https://www.engadget.com/2018/05/24/report-finds-android-malware-pre-installed-on-hundreds-of-phones/>. Acessado em 01/08/2018.
- Faruki, P., Bharmal, A., Laxmi, V., Ganmoor, V., Gaur, M. S., Conti, M. e Rajarajan, M. (2015). Android security: A survey of issues, malware penetration, and defenses. *IEEE Communications Surveys Tutorials*, 17(2):998–1022.
- Faruki, P., Ganmoor, V., Laxmi, V., Gaur, M. S. e Bharmal, A. (2013). Androsimilar: Robust statistical feature signature for android malware detection. Em *Proceedings of the 6th International Conference on Security of Information and Networks, SIN '13*, páginas 152–159, New York, NY, USA. ACM.
- Feng, P., Ma, J., Sun, C., Xu, X. e Ma, Y. (2018). A novel dynamic android malware detection system with ensemble learning. *IEEE Access*, 6:30996–31011.
- Genymotion (2020). Genymotion. <https://www.genymotion.com/>. Acessado em 11/02/2020.
- Gingo (2013). Android emulatordetector. <https://github.com/gingo/android-emulator-detector>. Acessado em 02/08/2018.
- Gupta, P. (2017). Decision trees in machine learning. <https://towardsdatascience.com/decision-trees-in-machine-learning-641b9c4e8052>. Acessado em 09/09/2018.
- GURUBARAN (2018). Android rat – thefatrat to hack and gain access to targeted android phone. <https://gbhackers.com/android-rat-kali-linux-tutorial/>. Acessado em 19/08/2018.

- Haase, A. (2017). A new era in mobile banking trojans. <https://securelist.com/a-new-era-in-mobile-banking-trojans/79198/>. Acessado em 19/08/2018.
- Hou, S., Saas, A., Chen, L. e Ye, Y. (2016). Deep4maldroid: A deep learning framework for android malware detection based on linux kernel system call graphs. Em *2016 IEEE/WIC/ACM International Conference on Web Intelligence Workshops (WIW)*, páginas 104–111.
- Jaiswal, M., Malik, Y. e Jaafar, F. (2018). Android gaming malware detection using system call analysis. Em *2018 6th International Symposium on Digital Forensic and Security (ISDFS)*, páginas 1–5.
- Kolbitsch, C., Comparetti, P., Kruegel, C., Kirda, E., Zhou, X.-y. e Wang, X. (2009). Effective and efficient malware detection at the end host. páginas 351–366.
- Krebs, B. (2017). Tech firms team up to take down ‘wirex’ android ddos botnet. <https://krebsonsecurity.com/tag/wirex-botnet/>. Acessado em 20/08/2018.
- Kubovič, O. (2018). Ransomware para android em 2017: Novas infiltrações e extorsões mais graves. <https://www.welivesecurity.com/br/2018/02/16/ransomware-para-android-em-2017/>. Acessado em 20/08/2018.
- Kumar, M. (2018). Dkfbotkit - first android botkit malware. <https://thehackernews.com/2012/03/dkfbotkit-first-android-botkit.html>. Acessado em 07/09/2018.
- Kural, O. E., Şahin, D. , Akleyek, S. e Kılıç, E. (2019). Permission weighting approaches in permission based android malware detection. Em *2019 4th International Conference on Computer Science and Engineering (UBMK)*, páginas 134–139.
- Lavado, T. (2019). Em 10 anos no brasil, android foi de 2 smartphones para sistema operacional dominante do mercado. <https://g1.globo.com/economia/tecnologia/noticia/2019/11/26/ha-10-anos-no-brasil-android-foi-de-2-smartphones-para-sistema-operacional-dominante-do-mercado.ghtml>. Acessado em 08/02/2020.
- Lindorfer, M., Neugschwandtner, M., Weichselbaum, L., Fratantonio, Y., v. d. Veen, V. e Platzer, C. (2014). Andrubis – 1,000,000 apps later: A view on current android malware behaviors. Em *2014 Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, páginas 3–17.
- LydeckerBlack (2016). Spade - android apk backdoor embedder. <https://gbhackers.com/android-rat-kali-linux-tutorial/>. Acessado em 19/08/2018.
- Malik, S. (2016). System call analysis of android malware families. *Indian Journal of Science and Technology*, 9.
- ManifestoAndroid (2018). Manifesto do aplicativo. <https://developer.android.com/guide/topics/manifest/manifest-intro?hl=pt-br>. Acessado em 30/08/2018.
- Markovskaya, A. (2017). Loapi — this trojan is hot! <https://www.kaspersky.com/blog/loapi-trojan/20510/>. Acessado em 01/09/2018.

- Markovskaya, A. (2018). Trojan para android é capaz até de gravar conversas. <https://www.kaspersky.com.br/blog/skygofree-smart-trojan/10014/>. Acessado em 19/08/2018.
- Mas'ud, M. Z., Sahib, S., Abdollah, M. F., Selamat, S. R., Yusof, R. e Ahmad, R. (2013). Profiling mobile malware behaviour through hybrid malware analysis approach. Em *2013 9th International Conference on Information Assurance and Security (IAS)*, páginas 78–84.
- Memon, A. M. e Anwar, A. (2015). Colluding apps: Tomorrow's mobile malware threat. *IEEE Security Privacy*, 13(6):77–81.
- O'Donnell, L. (2018). New banking trojan can launch overlay attacks on latest android versions. <https://threatpost.com/new-banking-trojan-can-launch-overlay-attacks-on-latest-android-versions/132858/>. Acessado em 02/08/2018.
- Pham, D.-P., Vu, D.-L. e Massacci, F. (2019). Mac-a-mal: macos malware analysis framework resistant to anti evasion techniques. *Journal of Computer Virology and Hacking Techniques*, páginas 1–9.
- pjlantz (2017). Droidbox. <https://github.com/pjlantz/droidbox>. Acessado em 11/09/2018.
- RAY, S. (2017). Understanding support vector machine algorithm from examples (along with code). <https://www.analyticsvidhya.com/blog/2017/09/understaing-support-vector-machine-example-code/>. Acessado em 08/09/2018.
- Rouse, M. (2017). Distributed denial of service (ddos) attack. <https://searchsecurity.techtarget.com/definition/distributed-denial-of-service-attack>. Acessado em 20/08/2018.
- Skygofree (2018). Skygofree — powerful android spyware discovered. <https://thehackernews.com/2018/01/android-spying-malware.html>. Acessado em 01/09/2018.
- Snow, J. (2016). Tech firms team up to take down 'wirex' android ddos botnet. <https://www.kaspersky.com/blog/dresscode-android-trojan/13219/>. Acessado em 20/08/2018.
- SRIVASTAVA, T. (2018). Introduction to k-nearest neighbors: Simplified (with implementation in python). <https://www.analyticsvidhya.com/blog/2018/03/introduction-k-neighbours-algorithm-clustering/>. Acessado em 08/09/2018.
- Statista (2018). Global market share held by smartphone operating systems from 2009 to 2017. <https://www.statista.com/statistics/263453/global-market-share-held-by-smartphone-operating-systems/>. Acessado em 02/10/2018.
- Strace (2019). strace - linux syscall tracer. <https://strace.io/>. Acessado em 06/11/2019.
- Symantec (2013). Android.obad. <https://www.symantec.com/security-center/writeup/2013-060411-4146-99>. Acessado em 19/08/2018.

- Symantec (2014). Android.samsapo. <https://www.symantec.com/security-center/writeup/2014-050111-1908-99>. Acessado em 19/08/2018.
- Tran, N., Nguyen, N., Ngo, Q. e Le, V. (2017). Towards malware detection in routers with c500-toolkit. Em *2017 5th International Conference on Information and Communication Technology (ICoIC7)*, páginas 1–5.
- Unuchek, R. (2013). An sms trojan with global ambitions. <https://securelist.com/an-sms-trojan-with-global-ambitions/59387/>. Acessado em 19/08/2018.
- Utku, A., DoGru, I. A. e Akcayol, M. A. (2018). Permission based android malware detection with multilayer perceptron. Em *2018 26th Signal Processing and Communications Applications Conference (SIU)*, páginas 1–4.
- VirusTotal (2019). Virustotal. <https://www.virustotal.com/pt/>. Acessado em 08/01/2019.
- Wisniewski, R. (2010). A tool for reverse engineering android apk files. <https://ibotpeaches.github.io/Apktool/>. Acessado em 21/08/2018.
- Wu, D., Mao, C., Wei, T., Lee, H. e Wu, K. (2012). Droidmat: Android malware detection through manifest and api calls tracing. Em *2012 Seventh Asia Joint Conference on Information Security*, páginas 62–69.
- Wu, D. J., Mao, C. H., Wei, T. E., Lee, H. M. e Wu, K. P. (2012). Droidmat: Android malware detection through manifest and api calls tracing. Em *2012 Seventh Asia Joint Conference on Information Security*, páginas 62–69.
- Wu, S., Zhang, Y., Jin, B. e Cao, W. (2017). Practical static analysis of detecting intent-based permission leakage in android application. Em *2017 IEEE 17th International Conference on Communication Technology (ICCT)*, páginas 1953–1957.
- Xu, K., Li, Y. e Deng, R. H. (2016). Iccdetector: Icc-based malware detection on android. *IEEE Transactions on Information Forensics and Security*, 11(6):1252–1264.
- Yerima, S. Y. e Sezer, S. (2018). Droidfusion: A novel multilevel classifier fusion approach for android malware detection. *IEEE Transactions on Cybernetics*, páginas 1–14.
- Zaman, M., Siddiqui, T., Amin, M. R. e Hossain, M. S. (2015). Malware detection in android by network traffic analysis. Em *2015 International Conference on Networking Systems and Security (NSysS)*, páginas 1–5.
- Zheng, M., Sun, M. e Lui, J. C. S. (2013). Droid analytics: A signature based analytic system to collect, extract, analyze and associate android malware. Em *Proceedings of the 2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TRUSTCOM '13*, páginas 163–171, Washington, DC, USA. IEEE Computer Society.
- Zhou, W., Zhou, Y., Jiang, X. e Ning, P. (2012). Detecting repackaged smartphone applications in third-party android marketplaces. Em *Proceedings of the Second ACM Conference on Data and Application Security and Privacy, CODASPY '12*, páginas 317–326, New York, NY, USA. ACM.

Şahin, D. , Kural, O. E., Akleyek, S. e Kiliç, E. (2018). New results on permission based static analysis for android malware. Em *2018 6th International Symposium on Digital Forensic and Security (ISDFS)*, páginas 1–4.